ALGORITHMS FOR MARS SPECTRAL CT

A thesis submitted in partial fulfilment of the
requirements for the Degree of
Master of Science in Medical Physics
at the
University of Canterbury
by
David Warwick Knight

University of Canterbury
2015

ABSTRACT

This thesis reports on algorithmic design and software development completed for the Medipix All Resolution System (MARS) multi-energy CT scanner. Two areas of research are presented - the speed and usability improvements made to the post-reconstruction material decomposition software; and the development of two algorithms designed for the implementation of a novel voxel system into the MARS image reconstruction chain.

The MARS MD software package is the primary material analysis tool used by members of the MARS group. The photon-processing ability of the MARS scanner is what makes material decomposition possible. MARS MD loads reconstructed images created after a scan and creates a new set of images, one for every individual material within the object. The software is capable of discriminating at least six different materials, plus air, within the object. A significant speed improvement to this program was attained by moving the code base from GNU Octave to MATLAB and applying well known optimisation routines, while the creation of a graphical user interface made the software more accessible and easy to use. The changes made to MARS MD represented a significant contribution to the productivity of the entire MARS group.

A drawback of the MARS image reconstruction chain is the time required to generate images of a scanned object. Compared to commercially available CT systems, the MARS system takes several orders of magnitude longer to do essentially the same job. With up to eight energy bins worth of data to consider during reconstruction, compared to a single energy bin in most commercial scanners, it is not surprising that there is a shortfall. A major performance limitation of the reconstruction process lies in the calculation of the small distances travelled by every detected photon within individual portions of the reconstruction volume. This thesis investigates a novel volume geometry that was developed by Prof. Phil Butler and Dr. Peter Renaud, and is designed to partially mitigate this time constraint. By treating the volume as a cylinder instead of a traditional cubic structure, the number of individual path length calculations can be drastically reduced. Two sets of algorithms are prototyped, coded in MATLAB, C++ and CUDA, and finally compared in terms of speed and visual accuracy.

The way the processor industry is going,
is to add more and more cores,
but nobody knows how to program those things.
I mean, two, yeah; four, not really; eight, forget it.

— Steve Jobs (n.d.). [1]

ACKNOWLEDGMENTS

I would like to thank Prof. Philip Butler, Assoc. Prof. Anthony Butler, Dr. Stephen Bell, Dr. Steve Marsh, Dr. Peter Renaud, Christopher Bateman, and Brian Goulter for supervising my master's research. Phil provided an endless supply of inspiration and refreshing optimism for everyone in the MARS group, and he, Anthony and Stephen always encouraged a friendly and approachable working environment. Thanks to Steve for running the Medical Physics program and for offering useful and timely advice throughout my journey. Many thanks to Christopher Bateman who gave me insight into the MARS project, particularly with his mind-bending explanations of the mathematics behind material decomposition. He also went above and beyond when it came to proof-reading this document. Special thanks to Brian who was my sounding board throughout this entire process. Brian could always find the time to listen to my crazy ideas and exhausting thought processes. He is also credited with creating many of the amazing diagrams littered throughout this thesis. I would also like to acknowledge the MARS group for awarding me a masters scholarship. The support given by everyone in the MARS group, of which there are far too many to name, was always much appreciated. Of special note are Kishore and Joe who eagerly embraced the new MARS MD, offered ideas for further improvements and finding all of the sneaky bugs. Thanks to Niels de Ruiter for his technical assistance with the MARS reconstruction workstation.

Finally I would like to express my love and appreciation to my family who supported me during my studies, particularly Danielle Barclay who has been my rock and never stopped believing in me.

CONTENTS

LIST OF FIGURES

1

INTRODUCTION

This thesis reports on developments I have made in spectral computed tomography (CT) image reconstruction and material decomposition (MD) techniques for the Medipix All Resolution System (MARS). The current MARS image processing chain does a good job, but there is room for significant improvement in terms of both computation speed and image quality. As at the end of this thesis, reconstruction can take anywhere from a couple of hours to a couple of days, depending on the size of the data sets being processed. Material decomposition is the second step in the image processing chain, with it taking several hours to process an entire set of CT slices. Because of these issues, research being conducted by students in the MARS group is often stifled, with the progress of the entire project being restricted as a flow-on effect. In order for the MARS scanner to become a commercially successful product, the performance of these key processes need to be improved. The development of hardware and software for the MARS project is a dynamic and fast-moving process, with new and improved systems always on the horizon. Creating efficient software is an important consideration, but so too is the type of hardware that the software will run on. The increasingly powerful graphics processing units (GPUs) offer potential solutions to the ever-present need for faster image reconstruction. But the answer is not so clear-cut, and the benefits and drawbacks of both the central processing unit (CPU) and the GPU must be carefully considered for every new algorithm. There is no one-size-fits-all piece of hardware that will satisfy all situations.

MARS MD is a software package that is an important clinical tool used by many members of the MARS research group. By analysing the reconstructed image slices taken by the MARS scanner, the software is able to distinguish individual materials within an object and produce a new set of images that display this information. Data sets can consist of well over 100 slices, which in turn adds up to several gigabytes of data needing to be processed. At the start of this research project, MARS MD was running on the Blue Fern supercomputer located at the University of Canterbury, and a typical data set would take between 5-12 hours to complete. Students and clinical researchers are regularly performing MD on their experimental data, and the task of improving the performance of MARS MD was the first objective of this research project. The software underwent many refinements, including making changes to the underlying data structures. Porting

the code base from GNU Octave to MATLAB allowed for a number of additional improvements, including parallelising the three core algorithms and designing a graphical user interface (GUI). The work completed on MARS MD for the first part of this thesis ultimately led to some impressive performance improvements, with computation times reducing from hours to minutes. This result has led to a significant increase in the volume of MD work being completed by students, and has contributed to the publication of several research papers and research grant applications.

The MARS image processing chain is made up of several discrete steps: (1) image pre-processing, such as ignoring bad pixels and reducing detector noise; (2) CT image reconstruction; and (3) material decomposition. A major area of research currently being conducted promises to merge these steps into a single operation. The new algebraic reconstruction technique (ART)-based reconstruction method is heavily reliant on the polychromatic Beer-Lambert Law, and is being developed by members of the MARS group. By performing image reconstruction and material decomposition simultaneously, the MARS scanner will be able to produce better quality images in a fraction of the time. Iterative image reconstruction methods such as ART are notoriously slow, in part because of the significant number of individual photon path lengths that need to be calculated at each iteration. One portion of the new algorithm relates to how the scanned object is represented in memory, with the object being treated as a cylinder instead of a traditional cuboid. The cylindrical volume is composed of concentric rings, and the individual voxels within each ring are a very close approximation to being cubic. Developed by Prof. Philip Butler and Dr. Peter Renaud (University of Canterbury), this novel approach to cylindrical volume geometry investigated in this thesis is capable of significantly reducing the number of photon path length calculations needed to complete a reconstruction. In a traditional cuboid reconstruction, the system matrix composed of the individual path lengths required per iteration is so large that it can't realistically be stored in memory. Instead, it must be repeatedly computed during the reconstruction process, a major factor in the poor performance of ART-based algorithms. The cylindrical volume geometry is capable of reducing the size of the system matrix by several orders of magnitude, allowing the system matrix to be pre-computed and stored before the scan takes place. The second part of this thesis presents programming implementations and analysis for a large part of the cylindrical volume geometry formulation, and provides valuable insight into what will become a core component of the MARS scanner.

The remainder of this chapter briefly discusses the rationale and significance of the MARS project. Section 1.1 outlines my key motivations for completing this research project; section 1.2 briefly discusses the clinical importance of the MARS project; and section 1.3 describes the

software and hardware tools employed throughout this thesis. The chapter concludes in section 1.4 with outlines of the chapters in this thesis.

1.1 RESEARCH MOTIVATION

With a background in both computer science and physics, I find myself fascinated by the interdisciplinary work being conducted in the field of medical physics. The MARS group is developing cutting-edge technologies that will assist in making remarkable breakthroughs in medicine, a truly inspiring goal. I have always had a keen interest in the semiconductor industry and its ability to follow Moore's Law in developing faster computer processors, and the move to multi-core CPUs and GPUs has ushered in new and exciting programming paradigms. These technologies are routinely implemented in medical devices such as CT and MRI scanners, due to the large amount of data processing that needs to be completed quickly. The MARS scanner is no exception and improvements need to be made to the underlying computational efficiency of almost every aspect of the project, not only for the benefit of research students, but also for customers and clinical researchers who rely on the MARS scanner.

The motivation of this research project is to make positive contributions to the productivity of the MARS group, and the objectives for this thesis can be grouped into two sections:

- Implement a solution that takes advantage of any inherent parallelism in the existing material decomposition software, and improve the overall usability of the software to encourage student comprehension and experimentation.
- Perform the initial implementation and analysis of a novel approach to calculating photon path length within a reconstruction volume. By designing algorithms for both the CPU and the GPU and comparing their performance, the appropriate type of hardware that best suits the specific problem can be determined.

1.2 CLINICAL SIGNIFICANCE

The MARS scanner is capable of simultaneous discrimination of bone, soft tissues, and high density materials that are present in contrast pharmaceuticals. The imaging technology will make a significant contribution to the work undertaken by clinicians and medical researchers. Current

MARS research includes locating regions of atherosclerosis plaque with gold contrast agents, and measuring drug delivery to tumours in mouse models [3]. The image reconstruction software and the material analysis techniques that are discussed in chapter 3 are what makes this research possible, and delivering this analysis quickly is of vital importance to everyone involved in the MARS project.

1.3 DEVELOPMENT TOOLS

The key similarities and differences when the two computer systems used throughout this thesis are outlined in Table 1. All software development was completed on system #1, and development was greatly simplified due to the two systems sharing a number of important properties such as GPU compute capability. Even though the GPU housed within system #2 was significantly more powerful, it still had the same underlying technology and capabilities as the GPU in system #1. This meant that the same code could be used on both GPUs with no changes necessary. System #2 is the primary image reconstruction computer used by members of the MARS group. This system was the best option for algorithm performance testing because it represented similar hardware and software configurations that future MARS reconstruction systems are likely to employ.

All algorithm development and performance testing took place within the MATLAB programming environment. Being an interpreted language, MATLAB can suffer from extremely poor performance compared to code written in a compiled language such as C and C++. Xcode and NVIDIA Nsight Eclipse Edition were jointly used to develop subroutines in C++ and CUDA, which were executed within MATLAB through the MEX interface.

1.4 THESIS OUTLINE

This thesis reports on parallelisation methods applied to two distinct parts of the MARS scanner: (1) the material decomposition software called MARS MD, presented in chapter 3; and (2) the novel approach to cylindrical volume geometry that will be integrated into the image processing chain, presented in chapters 4, 5 and 6. Chapter 2 provides the necessary background to the work presented in this thesis. Overviews of the relevant x-ray physics and parallelisation techniques are presented in chapters 3 and 5, respectively. Finally, a conclusion to the thesis is given in chapter 7. Detailed summaries of chapters 2-6 are provided below.

Table 1: Computer hardware used for the software development and performance testing aspects of this thesis.

	System #1 (Development)	**System #2 (Performance Testing)**
Operating System	OS X 10.10 / Windows 7	Windows 7
Processor	Intel Core i7 @ 2.3 GHz	Intel Core i7 @ 3.6 GHz
Number of Cores	4	4
Memory	8 GB	48 GB
GPU (NVIDIA)	GeForce GT 650M	GeForce GTX 670
Compute Capability	3.0	3.0
Memory	1 GB	4 GB
Memory Interface	GDDR5	GDDR5
Memory Bandwidth	80 GB/s	192 GB/s
CUDA Cores	384	1344
CUDA Version	6.5	6.5

Chapter 2 provides background to the technologies of computed tomography (CT) and spectral CT. MARS and the Medipix detector are introduced, specifically in relation to spectral imaging. The medical imaging concepts of FBP and ART are described, and the chapter concludes with a brief discussion of the exciting field of high-performance computing.

Chapter 3 begins by describing the process of material decomposition and how it relates to the MARS scanner. The unique contributions of MD to medical imaging and other industries are discussed. The performance improvements made to the material decomposition software MARS MD are explained, as well as the design choices made when developing the graphical user interface. Finally, the overall contribution these changes made to the research efficiency of the MARS group is briefly discussed.

Chapter 4 introduces the cylindrical volume geometry, a novel approach to modelling a reconstruction volume developed by Prof. Phil Butler and Dr. Peter Renaud (University of Canterbury). This thesis implements only the two-dimensional aspect of the geometry, so the more appropriate term *circular volume geometry* is used throughout this document. The geometry is compared to similar work completed by other researchers, before deriving the important equations that relate to both fan-beam and parallel-beam x-ray sources.

Chapter 5 discusses the motivation and choices made in the partial implementation of the circular volume geometry, specifically in regards to the calculations of voxels intersected and path lengths within voxels. Because the voxels are treated as being up to 10^3 times smaller than those used by a traditional CT scanner, the algorithm development works under the assumption that not every individual path length needs to be calculated. This potentially time-saving measure is investigated when the algorithm is developed in MATLAB before porting to C++ and CUDA.

Chapter 6 continues development of the circular volume geometry by considering how the irregularly-shaped voxels are mapped onto the square pixels of a computer screen. A simple mapping process is implemented for the purpose of determining to what extent image artefacts influence the final image. For this purpose, two distinct phantoms were reconstructed using a FBP method.

2

BACKGROUND

This chapter introduces the core concepts and technologies relevant to this thesis. Section 2.1 introduces the concept of computed tomography, highlighting the differences between standard CT, dual-energy CT, and spectral CT; section 2.2 provides the background and motivation behind the MARS project; section 2.3 gives an overview of the two most common image reconstruction methods, FBP and ART; and the chapter concludes with section 2.4 which offers an introduction to some of the most common high-performance computing technologies.

2.1 COMPUTED TOMOGRAPHY IMAGING

Computed tomography (CT) is an imaging modality common in both medical and industrial practices, and uses x-rays to generate images with high spatial resolution while employing fast scan times. Because x-rays are known to cause skin damage, malignancies, and other side effects, radiologists must find the balance between delivering large radiation doses to patients and obtaining good quality diagnostic images. The image reconstruction process is also computationally expensive and time consuming. Figure 1 demonstrates the differences between competing CT technologies: standard single-energy CT; dual-energy CT; and spectral CT.

The first CT scanners were used clinically in the 1970s and have undergone many iterations over the years [5]. The introduction of CT greatly reduced the need for exploratory surgery and provided detailed anatomy information to the physician. X-rays used for medical imaging are most commonly produced from cathode ray discharge tubes that accelerate a beam of free electrons over a potential difference in a vacuum, where they are directed toward a metal anode. The energy lost by the electrons being decelerated and stopped in the anode is released as a spectrum of x-rays that is directed through an object and toward a detector. Most CT scanners employ energy-integrating detectors that measure the combined attenuation from multiple x-ray interactions within an object. The x-ray tube and detector are located on opposite sides of a rotating gantry, allowing the object to be scanned at any angle. With all of the combined projection data the scanner reconstructs grayscale tomographic images of the body, possibly revealing the presence of cancers and a wide variety of other pathologies [5]. Because the signal produced by

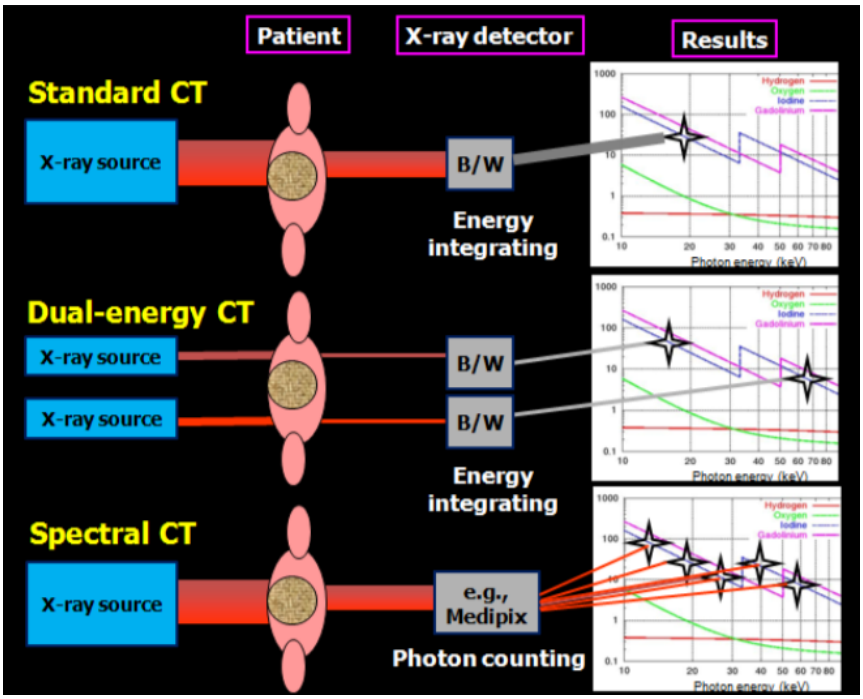


Figure 1: Differences between standard CT, dual-energy CT, and spectral CT. The photon counting capability of the Medipix detector is what differentiates the MARS scanner from competing dual-energy scanners [2].

the detector is proportional to the total energy deposited by all photons across the x-ray spectrum, any information about individual photons is completely lost. The subsequent image reconstruction process involves performing an iterative series of forward and filtered back-projection operations using these signals [6]. The grey levels in a CT image correspond to the level of x-ray attenuation, and standard CT scanners can not be reliably used to distinguish materials that have very similar attenuation properties.

Dual-energy CT scanners provide improved material separation by measuring the averaged flux over two separate x-ray spectra. This can be achieved in several ways, for example: using a single x-ray source that alternates between tube voltage; using alternating source filtrations; or using dual-source synchronous acquisition. The pair of attenuation measurements allow for the differentiation of two materials by exploiting the fact that the energy-dependent attenuation curves are different for distinct materials [2].

Spectral CT is the technique employed by the MARS scanner. It only requires a single spectrum and makes use of the polychromatic nature of the x-ray source. By using an energy discriminating detector it can measure the specific energy of every detected x-ray photon, allowing the scanner to group the entire x-ray spectrum into multiple energy ranges. By analysing the distribution of detected x-ray energies, it is possible to quantify the individual materials within the scanned

object, such as muscle, bone, fat, and contrast agents. The spatial resolution of standard CT is typically of the order of half a millimetre, but the MARS scanner can produce images with spatial resolution of the order of tens of microns, and is subsequently used for the evaluation of the micro-structure of specimens [2].

2.2 MARS

The Medipix All Resolution System (MARS) is a multi-energy CT scanner (Fig. 2) that employs the Medipix energy-discriminating detector. Developed at the European Organisation for Nuclear Research (CERN), these detectors are capable of acquiring a set of selected energy bins in parallel by processing each photon individually. The latest MARS scanners use the Medipix3RX detector.

Each energy bin is the summation of the individual photon counts that fall within a specific energy range [2]. These features are what makes the MARS scanner capable of generating spectral CT imagery, essentially allowing images to be created which can distinguish between clinically important materials that have similar attenuation properties, a feat not possible on modern CT scanners [7]. Spectroscopic photon-counting detectors can also increase the signal-to-noise ratio (SNR), thus allowing for a reduction in radiation dose to the patient. Current size limitations mean the scanner is only capable of accommodating small animals and excised samples of human tissue, but small animal CT imaging is used extensively for preclinical studies [8]. A human-sized scanner is planned for development within the next two years.

The MARS team is composed of members from many disciplines, including biologists, clinical radiologists, mathematicians, software engineers, electrical engineers, and physicists. The project is a collaboration between the universities of Canterbury and Otago.

The Medipix range of detectors have undergone several iterations following their initial introduction in 1999. From Medipix1 to Medipix3, the detectors have evolved from having an array of 64×64 pixels to an array of 256×256 pixels. The pixel areas have shrunk from 170×170 μm^2 down to 55×55 μm^2, and the number of energy thresholds has increased from one up to eight in the current generation Medipix3RX chips. Pixels can operate in one of two modes: (1) *single pixel mode* is where each pixel counts x-rays independently; and (2) *charge summing mode* is where communication occurs between neighbouring pixels, significantly improving the energy resolution [9].

Figure 2: Photograph of a MARS scanner.

There are a range of Medipix detectors, with the main discriminating factor often being the material that makes up the semiconductor sensor layer. A number of materials are used for this purpose, including Si, GaAs, CdTe, and CdZnTe [3]. The sensor layer is bump bonded to a Complementary Metal Oxide Semiconductor (CMOS) Application Specific Integrated Circuit (ASIC) readout layer. X-rays passing into the sensor layer produce an electron-hole cloud, which in turn causes a voltage pulse. The CMOS readout layer is able to analyse the pulse and calculate the energy of the x-ray that produced it [3].

MARS cameras are made up of vertically aligned Medipix3RX 14 mm $\times$ 14 mm detectors. In the current scanners, the source-to-detector distance (SDD) ranges from 100 to 250 mm. Future human-sized scanners will consist of about 100 detectors arranged in an arc.

2.3 IMAGE RECONSTRUCTION TECHNIQUES

2.3.1 *Background Physics*

When an x-ray beam passes through an object, its intensity decreases due to the interactions described in section 3.2. This process of attenuation is described by the Beer-Lambert Law, which

states that a logarithmic relationship exists involving: the intensity of an x-ray, I, passing through the object; the product of the distance, s, travelled by the x-ray; the x-ray energy, E; and the position and energy dependent linear attenuation coefficient of the substance, $\mu(E,s)$. The monochromatic form of the Beer-Lambert Law is typically used in its integral form

$$I(E) = I_0(E)e^{-\int \mu(E,s)ds} \tag{2.3.1}$$

where I_0 is the initial x-ray intensity and the linear attenuation coefficient is integrated over the entire path of the x-ray. To take full advantage of the Medipix chip in the MARS scanner, it is essential to consider the polychromatic form of the Beer-Lambert Law. Extending the monochromatic form by integrating over all energies of the polychromatic spectra found in the x-ray beam we obtain

$$\int I(E)dE = \int I_0(E)e^{-\int \mu(E,s)ds}dE \tag{2.3.2}$$

Within the object being scanned, different materials will attenuate the x-ray beam by different amounts. The linear attenuation coefficient characterises how likely an x-ray will be able to penetrate a substance, with a higher value representing a higher probability of attenuation within the substance and is a combination of all possible x-ray interactions. The overall coefficient for a composite object made up of m materials is the sum of all individual linear attenuation coefficients within the object,

$$\mu(E) = \sum_m \mu_m(E) \tag{2.3.3}$$

The linear attenuation coefficient of a substance (μ), with units of cm^{-1}, is directly proportional to the density so it is often convenient to instead refer to the mass attenuation coefficient (μ/ρ), with units of cm^2g^{-1}. The energy dependence of the mass attenuation for some common materials is shown in Fig. 3.

The sharp discontinuities observed in Fig. 3 are known as K-edges and occur at the binding energy of the K-shell electrons. Once an incoming photon has energy greater than the binding energy, there is a significant increase in the probability of photoelectric absorption and thus the attenuation coefficient undergoes a sudden five-fold increase as well. Every element has unique K-shell binding energies, many of which occur in the energy range of diagnostic x-rays. Exploiting these elemental "fingerprints" for diagnostic purposes is sometimes known as K-edge imaging.

Hounsfield units, also known as CT numbers, are a measure of attenuation often used in clinical CT systems instead of linear attenuation coefficients [3]. These units are scaled so that air has a

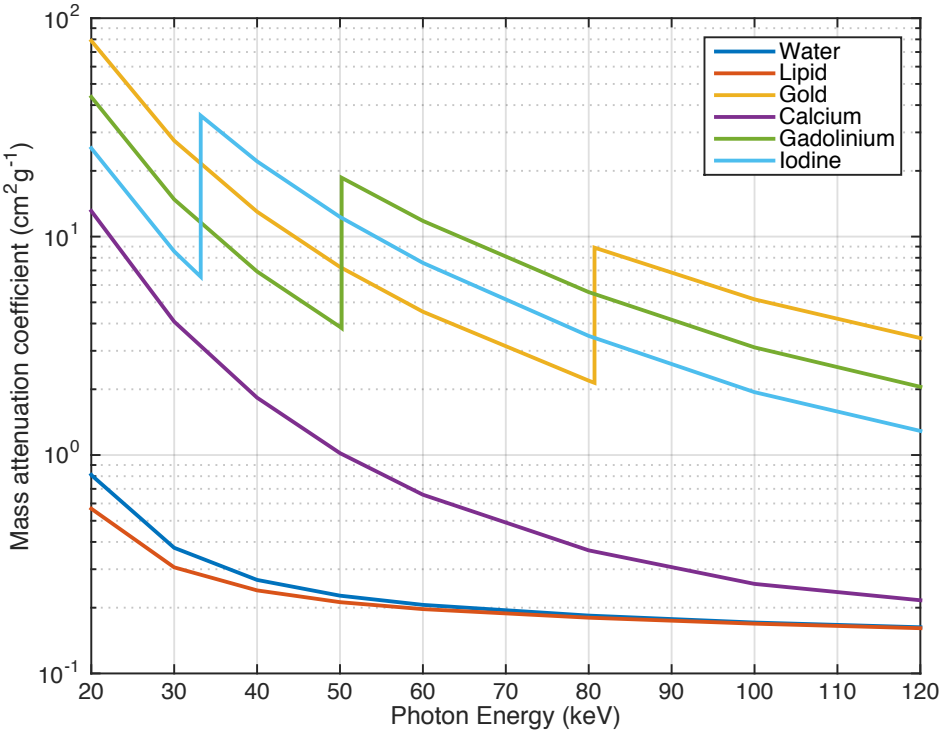


Figure 3: The MARS scanner collects photons in the range of 20 - 120 keV, and the mass attenuation profiles of some of the most commonly searched for materials are displayed. The human body is largely composed of lipid (fat), water and bone, which unfortunately have very similar attenuation profiles due to their K-edges sitting below the diagnostic imaging range [3].

value of -1000 and water has a value of zero. Hounsfield units for other materials are calculated with the equation

$$HU = \frac{\mu - \mu_{\text{water}}}{\mu_{\text{water}} - \mu_{\text{air}}} \times 1000 \tag{2.3.4}$$

Due to the energy dependence of linear attenuation, the associated Hounsfield unit for a material at different energies will also be different. To solve this problem for the purposes of spectral CT, the spectral Hounsfield unit was created [10].

Introduced in 1917 by Johann Radon and widely used in tomography, the Radon transform provides a mathematical model for the measured attenuation collected by the detector element. Taking Equation 2.3.1 and removing the energy dependence for simplicity, we can compute the integral of the attenuations as

$$\int \mu(s)ds \approx \sum \mu_i ds = \log \frac{I_0}{I} \tag{2.3.5}$$

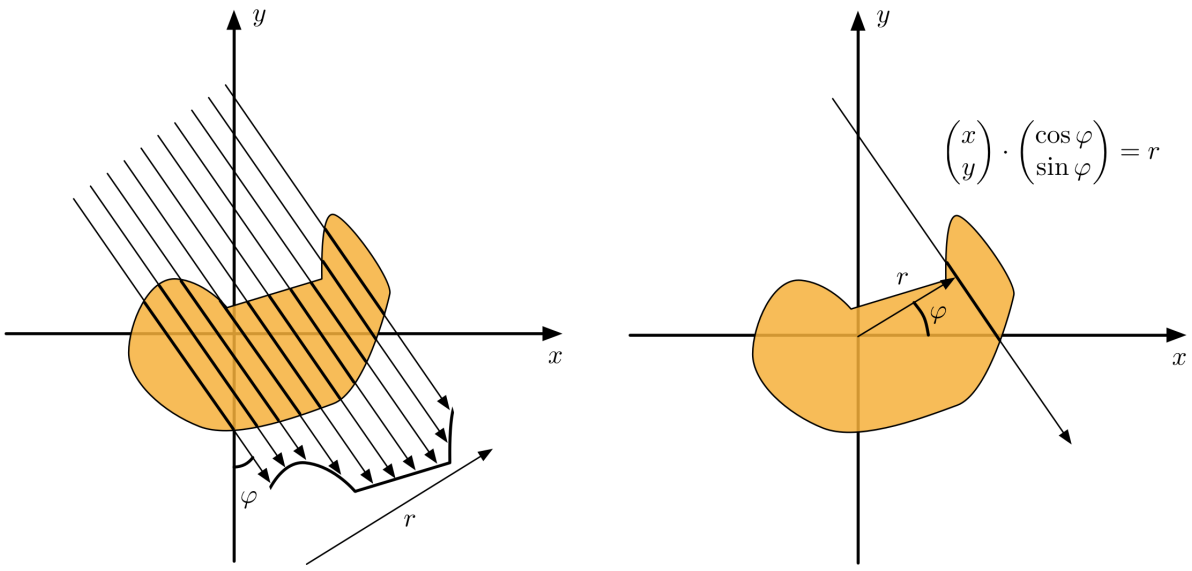
where the CT reconstruction procedure will attempt to derive the distribution of μ from the measured I_0/I.

A slice through the object being scanned can be treated mathematically as a distribution of attenuation coefficients in two dimensions, $\mu(x,y)$, and this function will be referred to as $f(x,y)$ for the remainder of this section. The x-ray beam is attenuated by every element of $f(x,y)$ along the line (Fig. 4b), so we formulate the equation

$$\begin{pmatrix} x \\ y \end{pmatrix} \cdot \begin{pmatrix} \cos\varphi \\ \sin\varphi \end{pmatrix} = x\cos\varphi + y\sin\varphi = r \tag{2.3.6}$$

Since there are many projection angles to consider, we derive the following parametrisation:

$$\begin{pmatrix} x(t) \\ y(t) \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix} \cdot \begin{pmatrix} r \\ t \end{pmatrix} = \begin{pmatrix} r\cos\varphi - t\sin\varphi \\ r\sin\varphi + t\cos\varphi \end{pmatrix}, \quad t \in \mathbb{R} \tag{2.3.7}$$



(a) Attenuation is a function of r (at a constant φ) and is measured at all rotation angles to generate the sinogram.

(b) An x-ray beam passing through a monotonic body. The orthogonal distance to the origin is r, and the equation of the line is $x\cos\varphi + y\sin\varphi = r$.

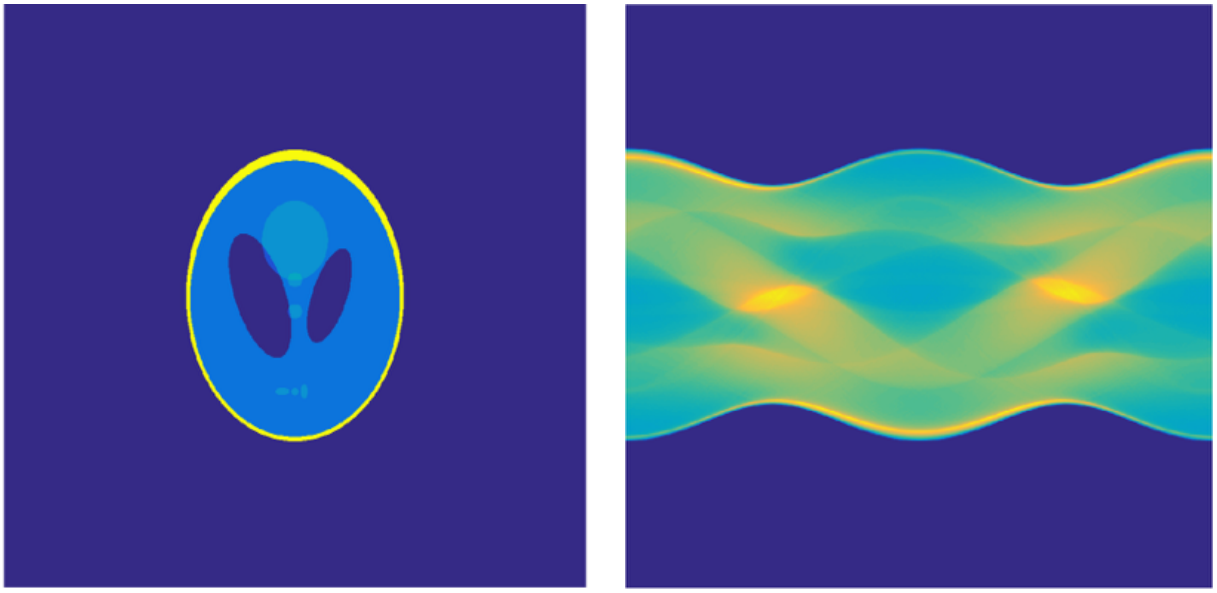
Figure 4: Coordinate system of the Radon transform.

where $\begin{pmatrix} r \\ t \end{pmatrix}$ is a line parallel to the y-axis, and the line is rotated through φ degrees using a rotation matrix. From Equations 2.3.5 and 2.3.7 the attenuation measured by the detector element is given by the line integral

$$\int f(x,y)ds = \int_{-\infty}^{\infty} f(r\cos\varphi - t\sin\varphi, r\sin\varphi + t\cos\varphi)dt \tag{2.3.8}$$

Finally, Equation 2.3.8 is referred to as the Radon transform, described by the formulation $f(x,y) \mapsto R_f(\varphi, r)$, where the left-hand side is defined in the plane $\mathbb{R}^2$ and the right-hand side is defined on $[0, 2\pi] \times \mathbb{R}$ [11].

Radon transform data is referred to as a *sinogram*, where the transform of small objects appear as blurred sine wave patterns with varying phases and amplitudes (Fig. 5). In terms of CT, the sinogram represents the radiographic projection data collected at all projection angles in a projection plane [3].



(a) Shepp-Logan phantom.

(b) Sinogram.

Figure 5: A Radon transform applied to a Shepp-Logan phantom generates a projection image known as a sinogram. Each column in the sinogram is made up all parallel projections for a given projection angle.

2.3.2 *Filtered Back-projection*

Filtered back-projection (FBP) algorithms employ inverse Fourier transforms and are commonly used in the reconstruction process of clinical scanners. However, these processes are based on analytical algorithms and don't perform well when the collected projection data is particularly noisy or under-sampled [12]. Back-projection is a relatively simple concept to understand. With a finite number of projections of an object contained in the sinogram, each projection can be smeared or back-projected along the lines they were measured from, resulting in an approximation of the original object. With no corrections applied, this simple method leaves significant blurring

in the reconstructed image. Filtered back-projection partially corrects this blurring by applying a high-pass filter to the projection. Commonly used filters are the ramp, Shepp-Logan, and Hann filters. The filter, g, is convolved with the raw projection data, $p(r,\varphi)$. The formulation for this procedure is given by

$$p_g(r,\varphi) = g \otimes p(r,\varphi) = \int_{-\infty}^{\infty} p(\ell,\varphi) g(r-\ell) d\ell \tag{2.3.9}$$

The image $\mu(x,y)$ can be calculated from the filtered projection data, $p_g(r,\varphi)$ using

$$\mu(x,y) = \mathcal{F}_{\mathbb{R}^2}^{-1} \left\{ \mathcal{F}_{\mathbb{R}^1} [p_g(r,\varphi)] \right\} \tag{2.3.10}$$

where the first step is to apply a Fourier transform to calculate the two-dimensional frequency spectrum of the image, and the second step is to apply an inverse Fourier transform of the image spectrum to obtain the reconstructed image.

2.3.3 *Algebraic Reconstruction Technique*

Algebraic reconstruction technique (ART) is a general term used to describe how to iteratively solve a large system of linear equations for the purposes of image reconstruction. Compared to FBP, iterative techniques generate higher-quality images, but are computationally more expensive. With recent advances in computing technologies such as GPUs, the implementation of ART for CT image reconstruction is becoming more commercially viable.

Treating the density distribution from the Radon transform as a system of linear equations, Sir Godfrey Hounsfield used an iterative reconstruction method in the world's first CT scanner [11] in 1972. An ART-based algorithm is employed by the MARS scanner.

Let the matrix R represent the Radon transform R_f seen in Equation 2.3.8. Every element in R stores the projection data for a given r and φ and we define $R(k,\ell) = R_f(k \cdot \Delta\varphi, \ell \cdot \Delta r)$. For example, $R(1,1) = R_f(0,0)$, $R(1,2) = R_f(0,\Delta r)$, $R(2,3) = R_f(\Delta\varphi, 2\Delta r)$, etc. The system of linear equations to be solved is formulated with the expression $Ax = b$, but to remain consistent with the subject matter, we will instead use $Sx = R$, where the matrix S is called the *system matrix* and describes the image formation process being conducted on the vector of volume elements, x. As described above, the measured data (Radon transform) from all projection angles is stored in R, re-formed as a vector made up of the matrix columns. S and R can be either real or complex.

With m angle steps and n parallel beams in the image reconstruction, we therefore have $m \times n$ linear equations in the unknowns x_i, where $i \in 1, \ldots n^2$ implies that all rays are organised in a column vector. Combining all of our knowledge of $Sx = R$, the following general form must be solved by the reconstruction:

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & \ldots & 1 & 0 & 0 & \ldots \\ 0 & 1 & 0 & \ldots & 0 & 1 & 0 & \ldots \\ 0 & 0 & 1 & \ldots & 0 & 0 & 1 & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{pmatrix}}_{S} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \ldots \\ x_{n^2} \end{pmatrix} = \begin{pmatrix} R(1,1) \\ R(1,2) \\ \ldots \\ R(1,n) \\ R(2,1) \\ \ldots \\ R(m,n) \end{pmatrix} \tag{2.3.11}$$

The matrix S has dimensions of $(m \times n) \times (n \times n)$. To account for photon path length through each individual voxel, weighting factors can also be applied to S to increase the accuracy at each projection angle. The general iterative method computes an approximation of the solution of the linear equations with the formula

$$x^{k+1} = x^k + \lambda_k \frac{R_i - \langle s_i, x^k \rangle}{||s_i||^2} s_i \tag{2.3.12}$$

where λ_k is a relaxation parameter, s_i is the i-th row of S, and R_i is the i-th component of R. Every voxel within x is iteratively updated until the reconstruction converges at a solution.

The traditional reconstruction model is composed of a cuboid that surrounds the scanned object, and is made up of thousands of smaller cubes or volume elements known as voxels. The term *voxel* is a combination of the words *volume* and *pixel*. Voxels are commonly used in the visualisation of medical data and for rendering three-dimensional computer games. One of the most time consuming parts of any iterative CT reconstruction algorithm is the calculation of photon path lengths through the millions of individual cuboid voxels within the object (Fig. 6).

The development of a new ART-based polychromatic reconstruction algorithm which takes full advantage of the multiple energy counters of the Medipix chip is being researched by members of the MARS group [3]. The cylindrical volume geometry investigated in this thesis is one part of the new reconstruction algorithm, with the intention of reducing the number of photon path length calculations needed during the iterative process. This in turn will reduce the time needed

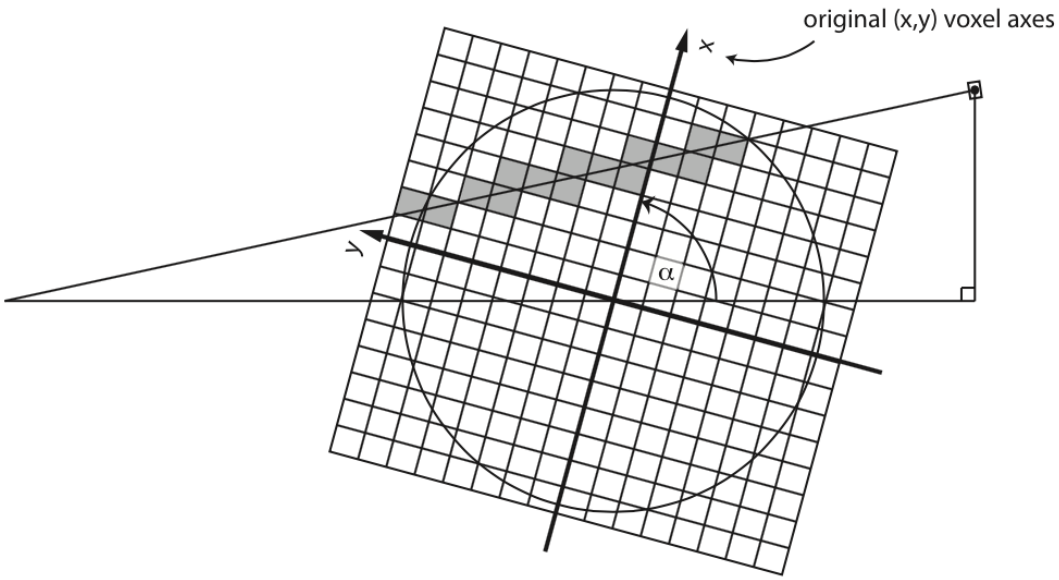


Figure 6: As the volume rotates about the z-axis, individual ray lines will pass through different combinations of voxels (MARS Bioimaging, 2014).

to compute the coefficient matrix, and will result in S becoming more sparse. Taking advantage of increased matrix sparsity is a major focus in developing the new algorithm.

2.4 HIGH-PERFORMANCE COMPUTING

The term *high-performance computing* (HPC) refers to how complex tasks can be accomplished efficiently by employing multiple processors or groups of computers [13]. HPC is often treated as a collective term that encompasses hardware and software systems, programming platforms, and parallel programming paradigms. Heterogeneous architectures, where the CPU and GPU work closely together, are leading the way in pushing new ideas and areas of research into parallel programming.

Many of the computational problems solvable by parallel computing operate on the principle that a large problem can be broken down into many smaller problems, each of which can be solved concurrently. In order to determine how a large problem can be mapped in this way, the programmer must consider the available computer architecture (hardware aspect) and the parallel programming model (software aspect).

The central processing unit (CPU) is the central component of any HPC system, and is often referred to as a *core*. To support parallelism at the architectural level, modern chip design in-

tegrates multiple cores onto a single processor, known as a *multi-core* processor. A parallelised computation essentially maps the segments of the large problem to the multiple cores which process parts of the solution concurrently. OpenMP (Open Multi-Processing) is a commonly used API (application programming interface) that allows a programmer to harness the multiple cores that reside in modern CPUs. It supports multi-platform shared memory multiprocessing memory programming in C, C++, and Fortran, and is compatible with most operating systems [14]. OpenMP consists of compiler directives, libraries and environment variables that can be easily incorporated into a subroutine to let it take advantage of multiple cores [15]. On a computer with N processor cores, the compiler will attempt to split a program containing OpenMP directives into N parts.

A program that can be divided up into a discrete series of calculations is called a *sequential program*. A *precedence restraint* (or a *data dependency*) describes the situation whereby the input of one calculation depends upon the output of a previous calculation. The sections of a program can be grouped in one of two ways: (1) some sections must be calculated after other sections due to a precedence restraint; and (2) some sections can be calculated at the same time as other sections because there is no precedence restraint. A program that contains sections of code that are computed at the same time as other sections is called a *parallel program*. Generally, a parallel program is made up of a series of sequential operations that are computed concurrently, but a program that contains many data dependencies is likely to restrict the use of parallelism.

The two fundamental types of parallelism are *task parallelism* and *data parallelism* [13]. Task parallelism refers to the situation where *functions* are distributed across multiple cores, and arises when multiple tasks need to be performed but can operate independently. Data parallelism refers to the situation where operations can be performed on multiple data items simultaneously, meaning that the *data* is distributed across multiple cores. This type of parallelism is what GPU are most suited for. GPUs were originally developed to assist with making computer games seem more realistic by allowing complex scenes to be quickly rendered on the computer screen. These types of calculations are generally known as SIMD (single instruction, multiple data), meaning that each pixel undergoes the same operation as all the others, but occurs using different data. Modern GPUs consist of a number of multi-processors and can process over a thousand instructions at once, making them ideal for SIMD algorithms. A similar concept is known as SPMD (single program, multiple data), whereby the same program is executed on multiple parts of the data [16]. This implies that each parallel unit may be executing different instructions, with some instructions being faster than others.

Each multiprocessor within a GPU contains a number of processor cores that are linked with very high bandwidth memory. While modern CPUs can manage thousands of threads, they can only run 4–12 at any given time, depending on the number of computational cores contained within the CPU [6]. The performance gain achievable on a GPU depends upon how easy it is to adapt a serial algorithm to run in a parallelised form. Thankfully, frameworks and tools are being developed and refined that allow such changes to be made in an efficient way. A popular framework is called the compute unified device architecture (CUDA). Developed by NVIDIA for use with their range of GPUs, it supports joint CPU/GPU application execution [17]. The term *compute capability* was coined by NVIDIA to describe the characteristics of its GPUs, and are composed of a major and a minor version number separated by a decimal point. The major version number refers to the class architecture and the minor version refers to minor differences within that class.

An alternative to CUDA is the open computing language (OpenCL). This language works on NVIDIA and non-NVIDIA GPUs alike, and is a good option if the software application is likely to be run on a wide range of different machines. Additional hardware possibilities also exist, for example Intel Xeon Phi coprocessors which are based on the Intel Many Integrated Core (MIC) architecture and offers up to 1.2 teraflops per coprocessor [18].

With future MARS scanners being scaled up to human size over the next two years, the amount of data needing to be rapidly processed will increase dramatically. For example, the current scanners can only accommodate small objects like mice. The size difference between a 20 g mouse and an 80 kg human is approximately 4×10^3, and therefore the computational complexity can be expected to increase by a similar magnitude. The need for more efficient image reconstruction algorithms is a necessity, with commonly adapted medical imaging algorithms to the GPU including image registration, image segmentation, and image de-noising. Real-time image de-noising can be achieved with GPU-based algorithms [6]. The scalability of parallelised algorithms running across multiple GPUs is the most cost efficient solution to such a problem, and finding the best way to implement these algorithms is an objective of this research project.

The algorithms in this thesis were developed using three techniques: (1) multi-core CPU implementation using MATLAB's Parallel Computing Toolbox; (2) single-core CPU implementation using C++; and (3) NVIDIA GPU implementation using CUDA. The first technique allowed for the initial prototypes of the algorithms, while the latter two allowed comparisons to be made in terms of speed and computational complexity, which will assist in building appropriate high-performance computing systems for future MARS scanners.

3

MARS MD

This chapter presents improvements made during this thesis to the material decomposition software, MARS MD, used by the MARS research team. By applying changes to data structures and algorithm implementation, and by using a parallelised structure, the speed of the program was improved by several orders of magnitude. Section 3.1 introduces some of the history and importance of the MARS MD software; section 3.2 describes the photon interactions in matter that are relevant to medical imaging; sections 3.3 and 3.4 give brief overviews of the material decomposition process and algorithm used in MARS MD; section 3.5 explains the core algorithm of MARS MD and the subsequent speed improvements; sections 3.6 and 3.7 describe the new user interface and additional features added to the MARS MD software; section 3.8 demonstrates some important research performed using the software; section 3.9 briefly discusses future developments to the software; and the chapter concludes with a summary in section 3.10.

3.1 INTRODUCTION

MARS MD is a post-reconstruction material decomposition program that is used to calculate the proportions of various materials in objects that have been scanned by a MARS system. Materials such as bone, tissue, gold, iodine, and contrast pharmaceuticals are routinely investigated in ongoing studies.

The internal algorithm design and prototype of MARS MD was created by Christopher Bateman (University of Otago) using MATLAB. The goal when designing a new algorithm is to first make it work as expected, with speed being a necessary oversight. It is common practice to design the flow of a new algorithm in a way that best suits the programmer, since debugging needs to be as simple as possible. There were several examples of poor coding practices within MARS MD, for example: the struct data type was used instead of arrays; memory was handled inefficiently; and the overall complexity meant that the end-user also had to be an expert-user in order to understand and use the software. The shortcomings of the software did not go unnoticed, and subsequently the code base was rewritten for the GNU Octave programming language in order for MARS MD to run as a parallelised process on the Blue Fern supercomputer. Octave is an open-

source project with a vibrant and dedicated community of developers and users alike. For many people, the primary reason for choosing to develop in Octave is the fact that it is completely free to use, unlike the moderately expensive MATLAB software package with its wide range of toolboxes. Both MATLAB and Octave are high-level interactive languages designed for performing numerical computations. Octave was created with MATLAB compatibility in mind and is able to natively run MATLAB m-functions [19]. The reverse is not true, however, since Octave adds additional syntax and functionality that MATLAB doesn't understand. Octave is primarily a command line tool and doesn't have a dedicated GUI, although some third party tools do exist. MATLAB is often the tool provided to students because of its easy to use GUI which makes the programming experience more interactive and improves the debugging experience considerably. MATLAB also comes with a powerful just-in-time (JIT) compiler which allows code to be compiled during execution of the program. This means that slow portions of code such as loops can be greatly improved by a process called vectorisation. Octave also supports a JIT compiler, but it is far less efficient than the one in MATLAB.

MARS MD is an important part of the MARS project because it enables specific materials to be identified and quantitatively measured from MARS scan data. Many of the astonishing images published in journals and student theses were created using the software, and some examples are presented in section 3.8. The work presented in this chapter describes improvements made to the MARS MD program.

The design goal for improving MARS MD was to make it simple for anyone to install and use, which would in turn allow pre-clinical research with the MARS scanner to be performed much faster. Compared to Octave, MATLAB is a more familiar programming environment for students, which was the primary reason to move the code base back to MATLAB. The entire MATLAB software package is provided to every student at the universities of Canterbury and Otago, where most MARS students are enrolled, thereby eliminating the cost benefit of using Octave. The secondary focus of rewriting MARS MD with MATLAB was to identify and fix inefficient portions of the code to improve computational time. The powerful toolboxes provided by MATLAB, particularly the Parallel Computing Toolbox, assisted with this task.

3.2 X-RAY INTERACTIONS IN MEDICAL IMAGING

The attenuation of an x-ray beam passing through an object can be caused by a number of fundamental interactions, three of which are relevant to diagnostic imaging and will be briefly described.

- Photoelectric absorption, due to the photoelectric effect, occurs when an incident photon interacts with an atom and leaves it in an excited state. An electron that was bound to the nucleus, known as a photoelectron, is ejected from the atom and leaves with kinetic energy equal to the energy lost by the photon. This effect relies on the condition that the x-ray energy is greater than the binding energy of the electron. The vacancy left by the missing photoelectron is filled by an electron in a higher shell, with the resulting energy difference either being lost by an ejection of a characteristic x-ray (fluorescent x-ray) or by another electron from a higher shell (Auger electron).
- Rayleigh scattering is caused by the oscillating electromagnetic field of a photon, which excites nearby electron to oscillate as well. This energy is released as scattered radiation, but it only contributes a very small amount to the overall attenuation of an x-ray beam within an object.
- Compton scattering treats both the x-ray and electron as particles which essentially collide like two billiard balls. Often the electron is treated as "free" but in reality it is likely located in an outer orbital of an atom. Both the photon and electron recoil in different directions with new energies. If the energy transferred is greater than the binding energy of the electron, the electron will be ejected from the atom.

3.3 MATERIAL DECOMPOSITION

The term *material decomposition* refers to the identification and classification of distinct materials within a volume. This technique was pioneered by Alvarez and Macovski in 1976 [20] who decomposed signals directly into the relative contributions of the photoelectric effect and Compton

scattering. This differs from methods used today whereby signals are decomposed directly into materials. The attenuation coefficient used by Alvarez and Macovski was given by

$$\mu(E) = C_{\text{photo}} \times \frac{1}{E^3} + C_{\text{compt}} \times f_{KN}(E) \tag{3.3.1}$$

where C_{photo} and C_{compt} are coefficients related to the cross-sections of the two interactions, and $f_{KN}(E)$ is known as the Klein-Nishina function which gives the differential cross-section of photons scattered from a single free electron. Equation 3.3.1 breaks down because the formulation of the photoelectric cross-section involves both Z and x-ray energy, and the power of Z depends on the photon energy. Alverez and Macovski instead treated the power of Z as a constant. It was also believed at the time that because there were only two significant x-ray interactions occurring in matter, it meant that only two materials without a K-edge could be identified, a statement shown to be false by Bornefalk who instead showed that the intrinsic dimensionality of the attenuation coefficient for low-Z materials is at least four in the diagnostic imaging region [21].

A photon that is counted by a multi-energy detector, such as the Medipix chip, will likely have passed through a number of different materials within the object being scanned. Building on the ideas from section 3.2, every material along the signal's path is modelled as a product of a basis function, $f_j(E)$, and a basis coefficient, a_j. At a specific energy, the overall attenuation coefficient of the signal, $\mu(E)$, is then composed of a linear combination of these material constituents,

$$\mu(E) = a_1 f_1(E) + a_2 f_2(E) + ... + a_N f_N(E) \tag{3.3.2}$$

The set of all basis functions is known before the decomposition begins. The goal of material decomposition is to find the set of basis coefficients that best solves the linear system. The MD algorithm can be generalised to be performed on both projection images and reconstructed images.

During a typical material decomposition procedure, there three types of mis-identification that can occur [3]. The first type is caused by large material basis sets because attenuation curves for different materials are often very similar, and trying to decompose materials in this situation leads to numerical instability. This problem was shown to be solveable by the work completed by Le and Molloi (2011) and Alessio *et al.* (2013) who developed algorithms that take advantage of sparse solutions to perform material identification and quantification as separate operations [22, 23]. The second type occurs when materials are deliberately omitted from the decomposition in an

attempt to reduce the number of unknowns, but instead this causes non-represented materials to appear as combinations of different materials. Liu *et al.* (2009) showed how an extra material can be decomposed with a dual-energy scanner by including mass fraction conservation constraints [24], and Wang *et al.* (2011) isolated materials with vastly different non-overlapping attenuation ranges by using segmentation techniques [25]. The third type results from using poor quality data containing image artefacts.

3.4 MD ALGORITHM

This section briefly outlines the MD algorithm used in the MARS MD software.

Reconstructed images are used to formulate effective mass attenuation coefficients for the chosen materials. Before decomposition begins, the data in each energy range was de-noised by taking the average of five adjacent images followed by a cylindrical median filter with a circle radius of one voxel. Next, the statistical segmentation function identifies low concentration high-Z materials as soft tissue if the signal is less than the level of noise present. These low concentrations are identified by modifying Euclidean norm thresholding with the Mahalanobis distance. Finally, the decomposition process employs the combinatorial material decomposition (CMD) algorithm developed by Christopher Bateman (University of Otago) for his PhD thesis, which uses 0-norm minimisation to achieve sparse solutions. The advantage of this algorithm is that it can deal with a problem set involving more materials than there are energies. It does this by calculating non-negative linear least squares solutions for a series of sub-problems, where fewer materials are considered at a time. The solution with the smallest least squares error out of all combinations is selected as the best solution. Therefore, unlikely or impossible solutions are rejected at an early stage [3].

3.5 SOFTWARE IMPROVEMENTS

A major problem with trying to solve such a large linear problem is that there are often too many unknowns to solve for all in one step. A solution is known as segmentation, which essentially assigns every reconstructed voxel to a specific subset of basis materials. In MARS MD, these subsets are: air, soft tissue, and dense materials with a high atomic number. By splitting up the image in this way, decomposition is able to be performed on each subset independently and

involves fewer unknowns to solve for at each iteration [3]. Taking a scan with a multi-energy detector allows for more than two energy ranges to be investigated, which in turn allows for more K-edges to be measured simultaneously.

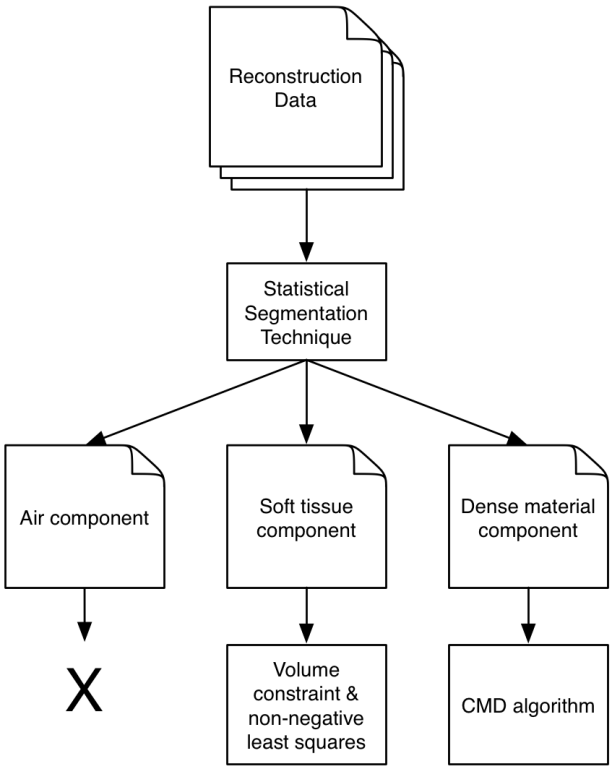


Figure 7: The core functionality of MARS MD. Air components are set to zero and ignored, soft tissue components are decomposed into lipid and water, and high-Z materials are decomposed with the CMD algorithm.

MARS MD is composed of three primary functions: median filter, segmentation, and material decomposition. Let there be m materials to be decomposed in the object, b energy bins, and let every image be made up of n^2 pixels. The MARS MD program has the following basic structure, with the main branches being displayed in Fig. 7:

1. Load covariance matrix ($b \times b$ total pixels).
2. Load energy data (reconstructed image) for a single slice for all energy bins and store as a multi-dimensional array ($b \times n^2$ total pixels).
3. Remove some of the noise from each data set with a median filter. This step is optional.
4. Create air, soft tissue, and high Z segmentation maps using the covariance matrix with the energy data sets ($3 \times n^2$ total pixels).

5. Perform a material decomposition using the segmentation maps and energy data sets, and store as a multi-dimensional array ($m \times n^2$ total pixels).

6. Write every individual material image to disk.

Using the MATLAB profiler (which measures execution time for individual functions within the code) to analyse the program it was clear that the processes in steps 3-5 were bottlenecks in the program, with each step corresponding to approximately one third of the running time. The MD algorithm has a high degree of parallelism because the processing of each pixel does not depend on the value of its neighbours. Pixels from one energy bin set only depended on pixels from the remaining energy bins that are situated in the exact same row and column.

Before any changes could be made to the MARS MD software within the MATLAB environment, the code base first had to be converted from Octave to the MATLAB programming language. Due to the inherent compatibility between the two platforms, this was a particularly simple task. For example, the most mundane yet sweeping alteration was changing all Octave friendly double quotes into the single quotes that MATLAB requires.

One of the best ways to speed up MATLAB code that consists of nested loops is called *vectorisation*. This process eliminates loops by replacing them with a single line of code that will perform the same operations. MATLAB is able to interpret what the code is trying to achieve and then execute it with its own highly-optimised array routines often at a far greater speed than the equivalent set of nested loops. Depending on how the loops work and what sort of calculations are being performed, vectorisation is often not possible or simply too difficult to achieve. Sometimes vectorisation of code will offer no speed benefits at all, due to the fact that the MATLAB JIT compiler can highly optimise loops making a negligible difference at run-time.

Before attempting to optimise array operations in MARS MD, the first step was to remove its heavy reliance on the struct data type. Each energy array was being loaded and stored as a separate field within a struct and being passed through the entire chain from one struct to the next. The struct data type has certain advantages, for example the ability to store more than one type of data within it, and also the ability to give each section of data a unique name that can be used to easily access it. This added functionality comes with extra overhead, and with there being no clear usage of these features in MARS MD, it was decided to instead store all data in a multi-dimensional array.

When performing operations on elements in a array, it's important to remember that MATLAB stores the values in column-major order. A remnant from the days of Fortran programming, con-

secutive elements in a column are stored in consecutive memory locations. The system will cache values along with their linear neighbours, so to enhance performance it makes sense to operate on a array column-by-column instead of row-by-row. In the case of a nested `for`-loop that operates over a square array, this means making the outer loop iterate over the columns and the inner loop iterate over the rows. Like many of the changes made to MARS MD, this one again offered small but noticeable speed improvements. The biggest improvement came as a follow-up to this one, where the Parallel Computing Toolbox was utilised and the outer `for`-loop was changed into a `parfor`-loop.

The concept of a `parfor`-loop is the same as for a `for`-loop, whereby the main program, also known as the client, will perform the statements contained within the loop. In the case of a `parfor`, the client will coordinate the execution of these loops with a series of workers contained within a parallel pool. By distributing sections of the data, referred to as slices, to the workers, the overall calculation can be performed in parallel and potentially run significantly faster than the standard loop. Once the workers have finished with their slice, the results are sent back and reassembled by the client. Every execution of a `parfor`-loop is called an iteration, with workers receiving their allocation of slices in no particular order. This is an important characteristic of the functioning of a `parfor`-loop, because in order for a speed improvement to occur each worker needs to be able to operate on its data completely separately from every other worker. Communication between workers, while certainly possible, is a large source of overhead and should be avoided whenever possible. MATLAB is very strict when it comes to what can and can't be performed within a `parfor`-loop, and often a great deal of work has to be put in to redesigning an algorithm to accommodate the strict rules. There are many reasons for why this type of loop may not be possible or desirable, such as if one iteration of the loop depends upon results from a previous operation, or if the communication cost between workers outweighs any performance benefit from running concurrently. It is not possible to use fields from a struct as sliced variables within a `parfor`-loop [26], which represents another reason to remove the program's reliance on this data type. There are workarounds for this behaviour, but they will again add unnecessary overhead to the function.

Some of the relevant variable types that may exist within a `parfor`-loop and are briefly described below.

- The loop variable is assigned by the initial `parfor` statement and defines the index value for each iteration of the loop. Loop variables must be a group of ascending consecutive integers and are used to coordinate the slices that are sent to workers.
- Sliced variables are those that are broken up and sent to workers. They can be either input or output data, or both. The client sends sliced input data to the workers and the workers send sliced output data back to the client.
- Broadcast variables are used within the loop but aren't affected by an assignment statement within the loop. These variables are usually transmitted to every worker individually, so a large broadcast variable will incur significant communication overhead between client and worker.

A parallel pool can exist as either a cluster of workers distributed over several machines on a network, or simply as a local pool operating on a desktop computer. Only the local parallel pool option was employed with MARS MD. In general, the maximum possible size of a local parallel pool equals the number of cores within the CPU of the computer. Some CPUs are capable of a technology known as hyper-threading, which essentially allows a single core to operate two processes at once. While hyper-threading may present more workers than there are available cores in the CPU, this generally won't translate into faster performance of a `parfor`-loop because hyper-threading doesn't offer any advantage when it comes to numerically intensive operations. MATLAB will ignore hyper-threading by default.

Each of the three bottlenecks in the code consisted of nested `for`-loops and are described below.

Median Filter

The original median filter function was made up of four nested `for`-loops, which is always an indication that speed improvements are possible. For every pixel in an energy array, an approximately circular shape is calculated around the pixel and the final value stored within the pixel is the median of those pixels surrounding it that lie within the circle. The two outer loops iterated over the rows and columns of pixels, but the two inner loops performed a similar iteration and were used to choose the pixels within each circle. While a simple idea, the set of four nested `for`-loops caused a major slowdown in the median filter operation. An alternative solution that eliminated the two inner loops was implemented and provided a significant speed improvement. Before the first outer loop began, a new square array consisting of ones and zeros was created

to act as a mask for the circular region of interest. The two inner loops are then replaced with vectorised code that tells MATLAB which square of values to pull out of the energy array. This new small array is then masked and the only values to emerge are those lying within the circular region. The median of these values is then taken as normal.

While this modification made a noticeable speed improvement, an even bigger improvement came simply by changing the outer-most `for`-loop into a `parfor`-loop. The function was already iterating in column-major order so copying data elements to the workers was already optimised in this regard. The only problem with this solution was the presence of a broadcast variable within the loop. Because each worker is required to read array elements from neighbouring columns within the array, this would have subjected the function to an additional communication overhead that will have caused the performance to suffer.

Segmentation

Within the segmentation function the Mahalanobis distance is computed several times, for example between each voxel to the lipid and water basis vectors. The Mahalanobis distance provides a measure for how many standard deviations away a measurement is from the mean of a multivariate normal distribution. In terms of the original MARS MD code, and ignoring the complex details, this involves multiplying the difference of basic vector points by the inverted covariance matrix. There are two problems with this method: the inverse of a matrix is primarily a theoretical value, and subsequently not an ideal choice when accuracy is required; and the matrix inversion process is slow. Because multiplying by the inverse of a matrix is performed as an alternate way of dividing by the original matrix, MATLAB provides an alternative syntax that is both faster and more accurate. Instead of multiplying by the inverse of a matrix, simply divide by the original matrix using the matrix right division (`/`) or matrix left division (`\`) operations. This improvement to the MARS MD code was incredibly minor and only offered a fractional improvement to the program's speed, but it is a good example of how important it is to take advantage of any optimised tools that are available.

In regards to the part of the function that performed the image segmentation, the only major change was again optimising the two nested `for`-loops into column-major order and changing the outer-loop into a `parfor`-loop. Similar to the median filter function, this loop was also forced to deal with a broadcast variable, but it generally only stored a small number of values, meaning

that the overhead would not have had a noticeable effect on performance.

Material Decomposition

This portion of the overall algorithm represents the culmination of the previous functions and generates a multi-dimensional array made up of one square array for every material being searched for in the object. It treats every pixel separately and allocates a value to it depending on whether it decides a material resides in that pixel. The same `parfor` conversion was applied here, and to avoid the use of large broadcast variables within the inner `for`-loop, temporary arrays were created at each iteration. Once an iteration was complete, the values stored in the temporary variables were transferred into the main output variables. After the MD process was completed, each section of the multi-dimensional array was written to disk as a TIFF image.

3.6 GRAPHICAL USER INTERFACE

For new members to the MARS research group and especially for those who were non-experts in the field of MD, the cumbersome command line instructions for running the program were particularly confusing and non-intuitive. A well designed graphical user interface enables a user to be more productive, which in turn benefits the entire team. Tools within MATLAB allowed the creation of a new user interface for MARS MD (Fig. 8).

Instead of the user having to manually type in the root folder path to the location of the scan data, the “Root folder” button in the GUI opens a familiar dialog box that performs the same task. The chosen folder is printed to its right giving a visual cue that the desired data sets were chosen correctly. The “Reload” button quickly reloads the configuration data from the last accessed location without the user having to search the file system again. These two buttons also allowed frequent and minor changes to the code base to be tested very quickly. Often when a decomposition fails to complete as expected, the problem lies in a mistake in the configuration file. For example, this could be as simple as a typo in the file or something more complicated like an inconsistent basis value for one of the materials. In the GUI design it was important to display all information found in the configuration file in an easy to read way so that the user was able to quickly check that all parameters were as expected before attempting the decomposition. The software detects the file types being loaded as either MAT, TIFF, or DICOM and displays this information to the user, where previously only the TIFF file type was supported. The scan data

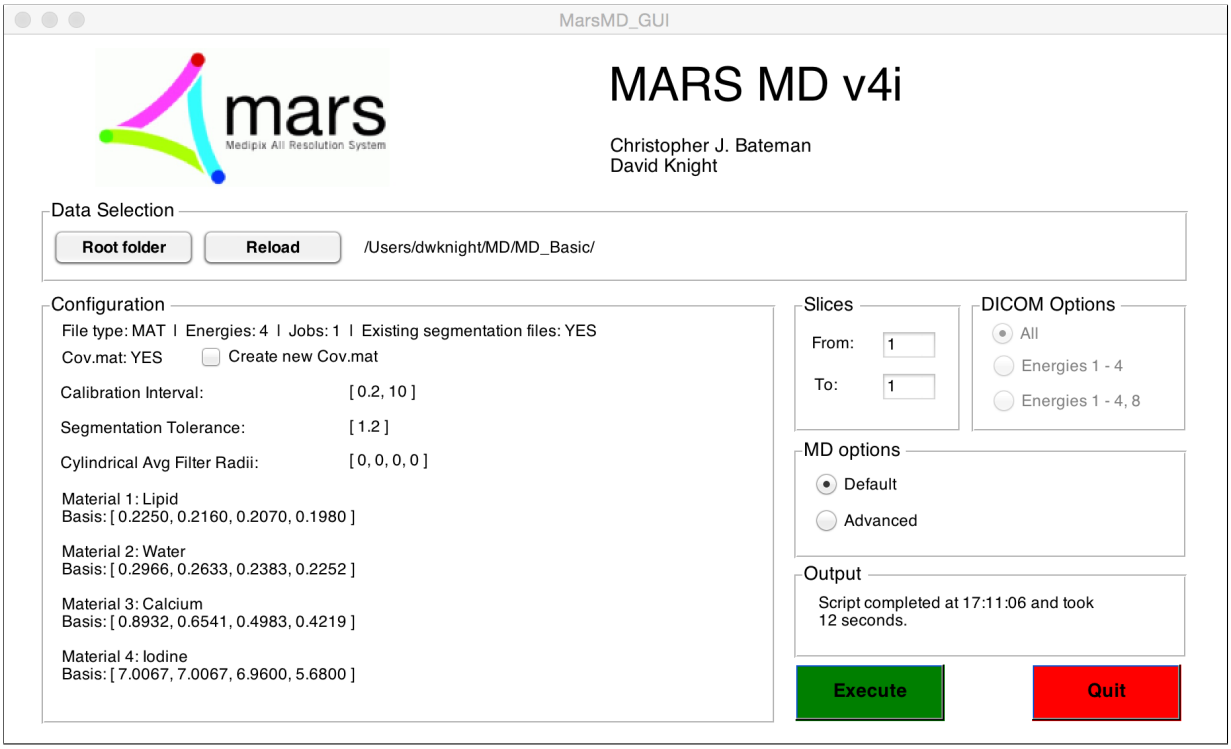


Figure 8: The MARS MD GUI was designed with non-expert users in mind.

is organised so that the images for each energy bin are stored in separate folders. The number of energy folders is listed, along with the number of slices (or jobs) that the software expects to be dealing with. The number of files in each folder is also checked for consistency. The “From” and “To” boxes are automatically filled in for the user, for example “From: 1” and “To: 135” for a folder of 135 slices. These values can be altered by the user, which can frequently occur when the beginning and ending slices contain no materials of interest. Useful feedback is an important aspect of any user interface, and the MARS MD GUI will display error messages if the user enters values that make no sense. For example, if a non-existent slice number is entered, or a starting slice number is higher than the ending slice number. If the root folder selected doesn’t fit the requirements needed to process the decomposition, an error message will suggest reasons for the inconsistency. When the user clicks the “Execute” button a progress bar is opened, displaying the number of slices remaining and the estimated time until completion. Under some circumstances the existing covariance matrix may be unsuitable, so if the user wishes to create a new covariance matrix for the decomposition there is a checkbox option available titled “Create new Cov.mat”. Once the “Execute” button is pressed, a dialog box appears prompting the user to enter the slice number that they wish to use to create the covariance matrix. After entering the slice number another window will appear that displays the slice and allows the user to draw a

polygonal region that is used to create the covariance matrix (Fig. 9). Once the polygon is closed the window disappears and the user is asked if they would like to save the new covariance matrix. Finally, the material decomposition continues as normal.

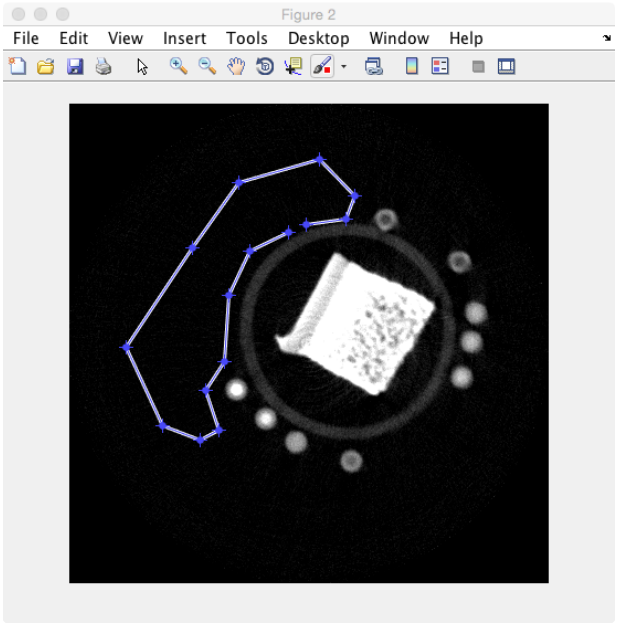


Figure 9: A popup window allows the user to draw a polygon around part of the CT image which is used to create the covariance matrix.

3.7 OTHER NEW FEATURES

Once the computation time had been reduced from hours to minutes and the new GUI had been completed, the software had become a very approachable and refreshing user experience. As a result of the software now being usable on a standard personal computer, significantly more decomposition work was being completed by MARS team members. The increased usability of MARS MD also led to a large number of bugs being reported and fixed, and many features that were requested by users were implemented and tested thoroughly.

Scan data sets ready for MD were originally in MAT format but had to be converted to TIFF due to the way Octave implemented certain 32-bit floating point images. With MATLAB being the primary platform again, this conversion was no longer necessary and the software was enhanced to allow the ability to load both MAT and TIFF files, depending on user preference. This was a welcome update, but the change didn't go far enough to truly streamline workflows, due to the fact that the MARS scanner now primarily works with the DICOM file format. Being the

recognised international standard, DICOM (Digital Imaging and Communications in Medicine) was developed in the 1990s in order to provide medical device manufacturers with a means of addressing technical interoperability issues. The standard allows for the integration of many different types of devices by defining how information is to be stored, printed, and transmitted through a network. The way data is stored in DICOM files is different to how it is stored in TIFF or MAT files. The DICOM standard describes how collected data is stored for many different types of imaging modalities. Because different modalities collect data in different number ranges, the pixel data for all modalities is scaled by a linear transformation into a limited range of numbers and stored to disk as a series of unsigned integers. For example, CT pixel values are measured in Hounsfield units, which can have negative values. To solve this problem, the DICOM file also stores a rescale intercept and a rescale slope value, which the DICOM viewer will use to convert the unsigned integer values back into they're original values. In order to support the loading of DICOM files in MARS MD, it was simply a matter of loading the rescale and intercept values from within the selected file and apply the linear transformation. A single DICOM file created by the MARS scanner can store up to 8 frames, each one corresponding to a different energy bin within the detector chip. The ability of MARS MD to load DICOM files also simplified the general decomposition workflow, since the software was now able to open a single file for every slice instead of several files. The program was modified to recognise when the user had selected a data set made up of DICOM files, and will then display the DICOM Options box on the screen (Fig. 8). This feature came at the request of users who wanted some level of control over which energy ranges stored within the DICOM file were used for the decomposition.

Because both developers and students were using the same piece of software, it became necessary to include a way for developers to try new algorithms without risk of breaking the existing code base. For this reason, the "MD Options" box was added to the GUI, essentially allowing any new algorithms to be "plugged in" without requiring any major changes to the software.

3.8 MARS MD GALLERY

Many students have produced publishable results and images that were made using the improved MARS MD software. Three of these results are briefly described below. All reconstructions were undertaken with a Medipix-3RX (CdTe) camera operating in charge-summing mode.

3.8.1 *Multi-material Phantom*

A multi-contrast phantom was created and filled with 12 polymerase chain reaction (PCR) tubes. Each of these capillaries contains varying concentrations of five materials [3]: gold, iodine, gadolinium, calcium and water. The PCR tube plastic was taken as a substitute for a sixth material, lipid. The exact concentrations are shown in Fig. 10a, and the phantom itself is a cylinder made from solid polymethyl-methacrylate (PMMA). The image reconstruction (Fig. 10b) was conducted by Dr Raja Aamir Younis (University of Otago).

During material decomposition of the multi-contrast phantom, the following material combinations were chosen: lipid and water; dense materials (calcium, gadolinium, gold, and iodine) with water; and the dense materials were also decomposed separately. The results are displayed in Figs. 10c and 10d and are generally encouraging. The water and dense materials were identified quite accurately, and some of the capillary walls made from PCR are correctly visible in the lipid image. Unfortunately, the low concentration of Au and Gd had signals within the level of noise, therefore the segmentation forced them to be identified as soft tissue.

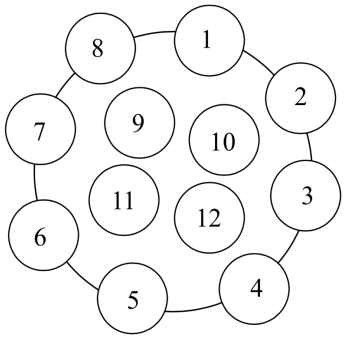
3.8.2 *Lamb Meat Imaging*

The image reconstruction of a small sample of lamb meat (Fig. 11) was conducted by Dr Raja Aamir Younis (University of Otago) and published by Aamir *et al.* (2014) [27].

Both the reconstructed image (top right) and the lipid material decomposition image (bottom right) have identified thin layers of fat in the muscle tissue [3].

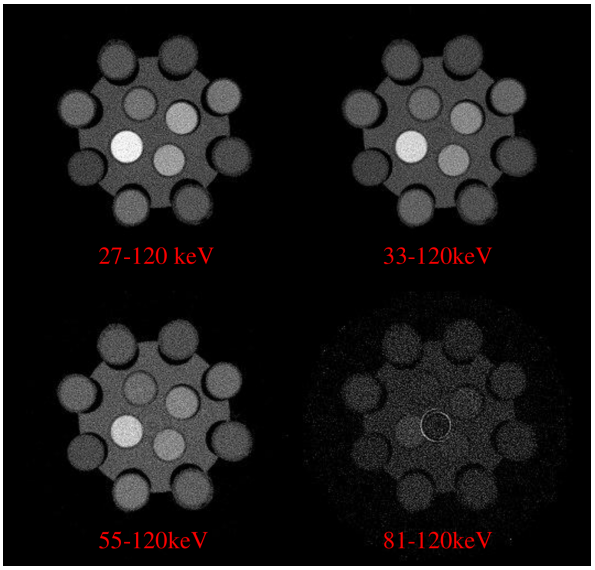
3.8.3 *Osteoporosis Arthritis Imaging*

Research is being conducted by several MARS team members to develop a non-invasive method for diagnosing the severity of osteoporosis arthritis within a patient. The investigation is a collaboration with the Christchurch Regenerative Medicine and Tissue Engineering (CReATE) Group, University of Otago, Christchurch, New Zealand. Excised cartilage samples are subjected to an iodine contrast (Hexabrix) before being scanned. The 3D image shown were created using the MARS visualisation software developed by Alex Chernoglazov (University of Canterbury).

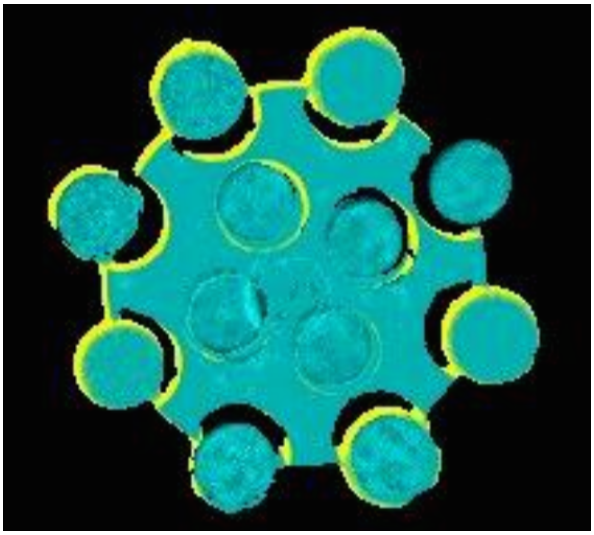


1 - 2 mg(Au)/ml
2 - 140 mg(Ca)/ml
3 - 2.1 mg(Gd)/ml
4 - 4.2 mg(Gd)/ml
5 - 8.4 mg(Gd)/ml
6 - Water
7 - 8 mg(Au)/ml
8 - 4 mg(Au)/ml
9 - 9 mg(I)/ml
10 - 280 mg(Ca)/ml
11 - 36 mg(I)/ml
12 - 18 mg(I)/ml

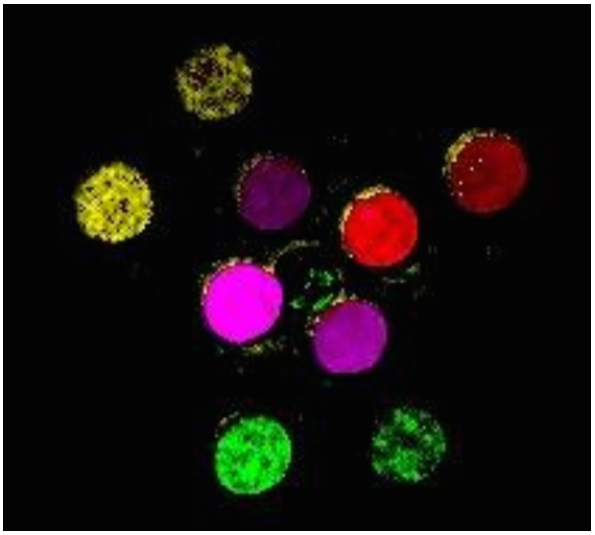
(a) Layout of the multi-contrast phantom showing the relative position and material concentration of each capillary.



(b) Each of the four energy bands on the multi-contrast phantom (slice 12) were reconstructed independently with ART.



(c) Soft tissue image containing lipid (yellow) and water (cyan).



(d) Dense material image containing calcium (red), gadolinium (green), gold (yellow), and iodine (magenta).

Figure 10: Multi-material phantom scanned with a Medipix-3RX (CdTe) camera operating in charge-summing mode [3].

The data was collected by Kishore Rajendran (University of Otago) and Caroline Löbker (University of Twente, Netherlands). The material decomposition images (Fig. 12) show that the iodine contrast has penetrated into the cartilage and indicates a severe amount of arthritis is present in the sample.

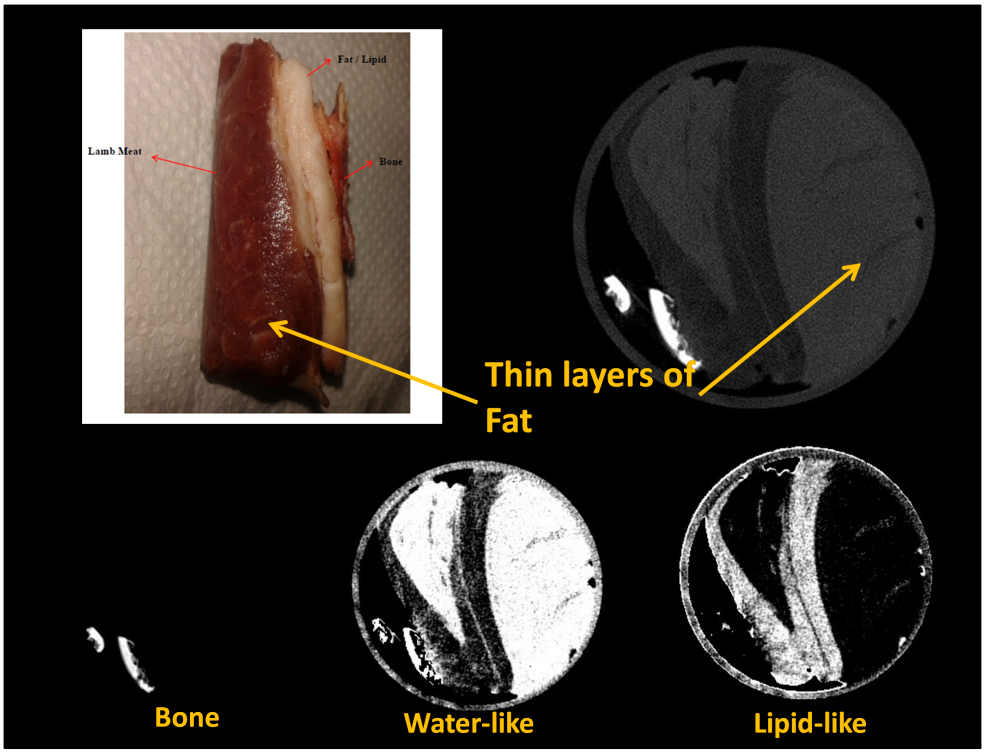
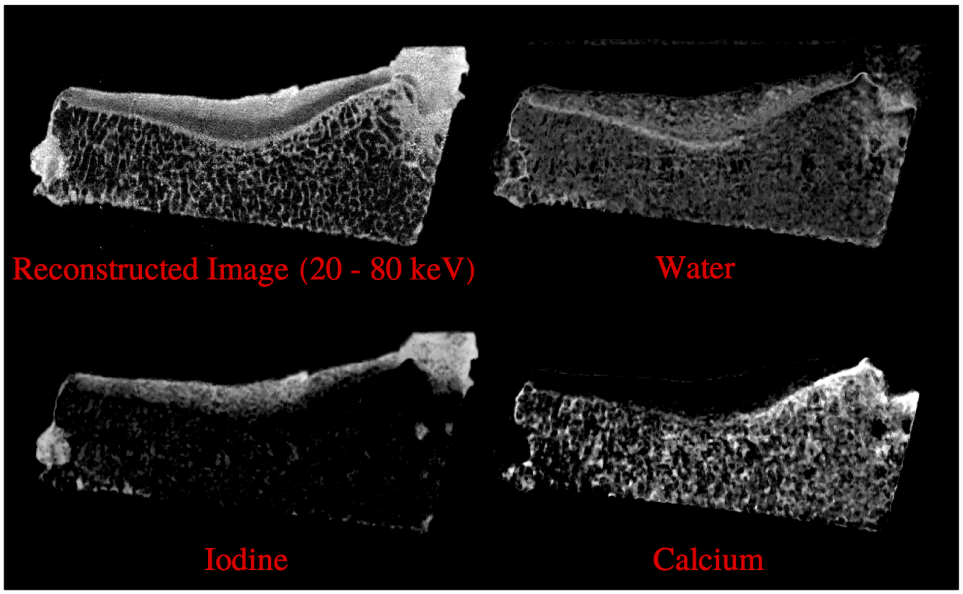


Figure 11: Photograph of the lamb meat sample (top left), reconstructed Hounsfield unit image of the sample (top right), and bone, water-like, and lipid-like material decomposition images of the same (bottom). [3].

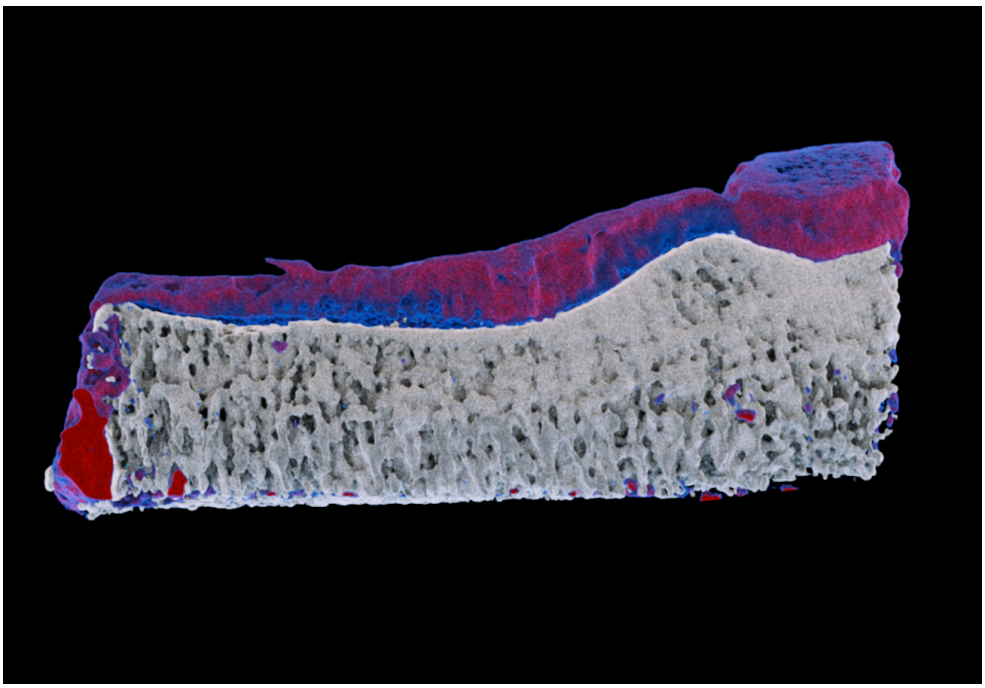
3.9 DISCUSSION

The Octave version of MARS MD running with 32 cores on the Blue Fern supercomputer took several hours to perform MD on a standard sized data set, compared to the MATLAB version which took just minutes while running on a desktop computer with 4 cores. The Octave version has long since been retired in favour of the MATLAB version, so it is difficult to determine why the performance was so poor. Designing software that works efficiently on a massively-parallel system such as Blue Fern is no simple task, and careful consideration and planning must be made to achieve a good result.

There is still work to be done on the MARS MD software. For example, there needs to be consideration given to how material information is properly stored in the DICOM tags, as this isn't a feature offered by default in the DICOM standard. Additionally, the entire MARS MD algorithm was ported to C++ for inclusion in a forthcoming image processing chain. Even though the C++ version was significantly faster, the MATLAB version is currently still favoured since GUI creation in MATLAB is very easy, and MATLAB scripts are cross-platform. The next phase of development



(a) The reconstructed Hounsfield unit image (top left) of the cartilage sample and the resultant MD images for water, iodine, and calcium.



(b) False colour 3D image showing a region of severe arthritis, indicated by iodine contrast (purple) almost reaching the bone layer of the tibial plateau.

Figure 12: Excised cartilage sample containing an iodine contrast agent. Arthritic cartilage is identified by deep penetration of the contrast agent [3].

will focus entirely on the C++ version, with the intention of integrating it into the MARS image reconstruction chain.

3.10 SUMMARY

- This chapter has presented work done during this thesis to provide significant improvements to the MARS MD software used by members of the MARS group. As a result of the changes made, the overall running time of the material decomposition algorithm has been reduced from hours to minutes, which led to a much higher throughput of MD research being completed.
- The history and background theory to the material decomposition method was described.
- The relevant performance-enhancement methods that MATLAB provides were briefly described, including the MATLAB profiler, vectorisation, making use of the Parallel Computing Toolbox, and understanding the underlying data structures.
- The three main functions of MARS MD: the median filter, segmentation, and material decomposition were briefly explained.
- Changes to the MARS MD algorithm include: moving the code base from GNU Octave to MATLAB; improving the way data is stored and manipulated; applying parallelisation methods to the three core functions with `parfor`-loops.
- A new GUI was designed to eliminate the complicated command-line interface of the original MARS MD, and resulted in the software becoming much more user friendly.
- Several other new features were added to MARS MD, including: DICOM import/export functionality; conversion utilities between MAT and TIFF files; and a special mode to allow expert users to plug in and test new algorithms with the MD processing chain.
- Three sets of research that used the improved MARS MD software were presented in a gallery. The material decomposition images shown have been published in journals, and featured in oral presentations and on posters produced by MARS team members.

4

CYLINDRICAL VOLUME GEOMETRY

This chapter introduces and formulates a novel variation of the reconstruction concept known as the cylindrical volume geometry. Developed by Prof. Phil Butler and Dr. Peter Renaud (University of Canterbury), it has the potential to significantly decrease the number of voxel-based calculations necessary during MARS image reconstruction. An advanced portion of the geometry formulation describes how a system matrix can be derived that is independent of the volume rotation angle. For simplicity, this aspect was not explored as part of this thesis.

Sections 4.1 and 4.2 discuss relevant background theory and related research to the algorithm development described in this chapter; sections 4.3, 4.4 and 4.5 discuss how the geometry is arranged and derive the required equations that describe it; section 4.6 discusses the novel ideas behind how image reconstruction can be sped up by determining x-ray path length in a faster and more approximate way; section 4.7 briefly describes two small algorithms necessary for integrating the geometry into a reconstruction program; and the chapter ends with a conclusion in section 4.8 and a summary in section 4.9.

4.1 INTRODUCTION

Image reconstruction using ART is time-consuming because the system matrix is generally too large to be stored in memory and therefore needs to be calculated on the fly. This is the reason why commercial CT scanners use the FBP method instead. Iterative reconstruction algorithms are composed of alternating forward and backward projections through the volume, where the forward projection is concerned with simulating measured data based on the current values within the system matrix. One part of the forward projection relies on the calculation of x-ray path length through each voxel, and traditional reconstruction methods calculate these lengths through a cuboid made up of cuboid voxels. Unfortunately, there is not enough symmetry in this geometry to provide any meaningful reduction in the required memory. The cylindrical volume geometry improves upon this problem by matching the geometry of the reconstruction volume to the geometry of the rotating gantry (Fig. 13), which reduces the number of geometric dimensions from

three to two due to circular symmetry. This thesis only investigates part of the novel approach being investigated in this thesis is only part of the formulation.

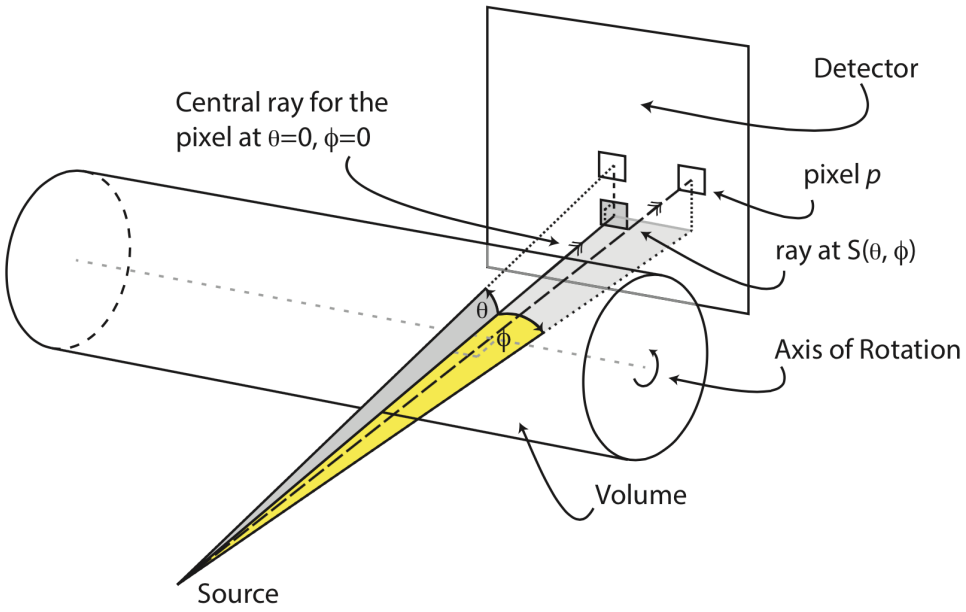


Figure 13: Right-handed set of Cartesian coordinates are fixed to the gantry frame and the cylindrical volume rotates and translates within the frame. The central ray is measured by the grey detector pixel, and rays at angles θ and ϕ are measured by nearby pixels (MARS Bioimaging, 2014).

In terms of the photon-counting detector in a MARS scanner, we must also consider the materials that might exist in each voxel, and the energies bins used to measure x-rays. The dimensionality of the forward projection has thus been reduced from five dimensions to four dimensions. Volume geometries that match the symmetry of the rotating gantry is not a new idea, but the novel ideas investigated in this thesis can be described in two parts:

1. The cylindrical volume is sliced into disks, with each disk being made up of concentric rings containing voxels that are approximately cuboid. The x-ray path is calculated on a ring-by-ring basis, significantly reducing the number of calculations needed per ray passing through the object.

2. The individual path lengths of every voxel are never calculated. Instead, the path length through each ring is calculated, and is then evenly divided amongst the voxels that have had the heaviest x-ray intersection.

With cone-beam CT systems like the MARS scanner, the detector pixels will be counting x-rays at angles ranging between $\pm\theta$ and $\pm\phi$ (Fig. 13). The short SDD of the current MARS scanner and

the vertical arrangement of Medipix detectors means that ϕ can vary between $\pm 5°$, and θ between $\pm 20°$. Future human scale MARS scanners will require a significantly larger SDD of approximately 2 m. This will have minimal effect on the range of θ, but the range of ϕ will be smaller.

We can make a good estimation of how many individual measurements are taken during a typical MARS spiral scan. Three Medipix3RX detectors together account for 48k pixels, each with eight counters per image frame. Furthermore, if the scanner collects data at 1500 angles for every 10 full turns, the combined total accounts for up to 5.9×10^9 measurements per scan. This large number of measurements translates directly into the computation time needed for an iterative reconstruction algorithm. Each of the 48k pixels requires x-ray paths be traced back through the reconstruction volume in order to determine how far each x-ray travelled in every voxel. The cylindrical volume geometry described in this chapter is able to significantly reduce the number of path length calculations required by employing circular symmetry and making small assumptions in regards to voxels and their neighbours. The entire system matrix will be created before the scan takes place, eliminating one of the most computationally expensive parts of the iterative algorithm and making a substantial improvement to the image reconstruction running time.

4.2 RELATED WORKS

There have been many other investigations into exploiting rotational symmetries and computing and storing a system matrix, with some of the more critical works described in this chapter.

Rodríguez-Alvarez *et al.* (2011) presented research where system matrices were constructed in the fastest possible way on a standard desktop PC. They note that exploiting a polar grid is often used in commercial CT scanners when preserving the spatial resolution of the scanner is not an essential requirement, but their methods instead aim to preserve the spatial resolution. Their methods were then upscaled to 3D and analysed in terms of system matrix size, computation time, and reconstructed image quality [28]. The creation of their system matrices were based on a polar coordinate system and took advantage of rotation symmetries of CT devices, and they implement several methods to achieve this goal. They describe how a 2D circular slice can be split up into rings containing v identical segments, with each ring containing a set of either circular or polar pixels. This layout allows for a reduction of system matrix size by taking advantage of circular symmetry. The amount of work needed to take advantage of only four symmetries might

not be worth the overall reduction in system matrix size. The authors discussed several methods of constructing circular volumes, and two of them are discussed in this section.

The first method is called Unitary Relation Aspect (URA) and maintains a unitary aspect ratio between Δr and $\Delta\theta$, the ring thickness and segment angle, respectively. This method results in "holes" in the field-of-view (FOV), but greatly simplifies the system matrix construction time. In order to preserve the spatial resolution of the scanner, the voxels are designed to have a diameter less than or equal to half of the spatial resolution of the scanner. The second method is called Constant Radius (CR) and most closely resembles the geometry described in section 4.3. This method uses a constant Δr to described the voxels, and the length of Δr is less than or equal to half of the spatial resolution of the scanner. The size of $\Delta\theta$ is decided in a similar manner.

The authors characterised the system matrices by a number of factors, including size and time of creation. The URA-C system matrix was 32 Mb in size and took 4 seconds to create, whereas the CR-P system matrix was a similar 40 Mb in size, but took 5 hours and 47 minutes to create. The time differences between these two methods is significant, and demonstrates how an increased complexity at the voxel level causes the overall complexity of the system matrix to increase by many orders of magnitude. Their polar system matrix sizes were 800 times less than those of Cartesian system matrices. The authors also analysed the image quality of reconstructions using these system matrices and is discussed in chapter 6. Their results showed that voxel arrangement on a polar grid causes variable resolution and oversampling of the central area of the FOV, meaning that the resolution of the central voxels was higher than necessary. In general, they showed that the reduction in system matrix size due to symmetry caused an increase in reconstruction efficiency without loss of quality. The speed at which the URA-C system matrix was generated indicates that a stored system matrix may not be necessary at all, and the voxel data can be created on the fly without a reduction in reconstruction speed.

As a means of reducing the computation time of a standard ART algorithm when dealing with noisy or incomplete projection data sets, Jian *et al.* developed a new algorithm which they named Polar-ART (PART) [29]. The key difference between the two algorithms is the rotating polar-coordinate system that is implemented in PART. The motivation for developing this method came about as a means to avoid using stored system matrices or look-up tables. The huge memory requirements for look-up tables can easily be several gigabytes or more, and placing such a heavy load on a standard desktop PC is not an ideal solution. The PART algorithm aims to create a middle-ground between a large look-up table, and having to compute the system matrix during reconstruction. The circular volume is sliced through the origin into evenly spaced segments

given it a high degree of symmetry, and only the projections through one segment of the volume needing to be calculated. A simple offset can be applied for each remaining segment. It was noted that memory requirements decreased by a factor of 600, and the rotating polar-coordinate techniques can be easily introduced into statistical iterative algorithms such as MLEM. The authors claim that PART is 2.7 times faster than ART and offers reduced memory requirements. They found that ART generates sharper edges than PART, but PART produces smoother images than ART. Overall, PART suffers from a loss of resolution and reduced quantitative accuracy in the reconstructed images.

The final symmetric-polar coordinate scheme is a more recent study performed by Rodríguez-Alvarez *et al.* (2013) and it builds on the voxel research that they conducted in their earlier publication. The research describes a method to reduce the computational cost for expectation maximisation iterative algorithms [8]. The authors claim that the entire 3D volume can be treated as a single entity, without the need to divide it into 2D slices. The weights matrix is first considered in 2D using three strategies: (1) nearest neighbour, where voxels only contribute to the attenuation of the nearest x-ray beam; (2) Joseph's method, where voxels contribute to its two surrounding beams, with weighting factors decreasing linearly with distance, and (3) intersected area, where weights are calculated based on an area that resides between detector element and x-ray source. The final method provided the best results and was scaled up to 3D with the addition of a third cylindrical coordinate.

Using the intersected area method in 3D the weight element for every voxel was calculated. Every x-ray beam was shaped like a pyramid, with the square detector element acting as the pyramid base, and the x-ray point source acting as the pyramid peak. The authors characterised the system matrices by a number of factors, including size and time of creation. A 3D Cartesian system matrix made up of $192 \times 192 \times 175$ voxels and using 200 projections, the size of the matrix was 8.5 GB and took 630 seconds to generate. In comparison, a 3D polar matrix made up of $204 \times 204 \times 175$ voxels and using 200 projections, the size of the matrix was only 224 MB and took 12 seconds to generate. The reconstruction quality was also tested, and these results are discussed in chapter 6.

Like other CT scanners, the MARS scanner rotating gantry has cylindrical symmetry. By fixing a Cartesian coordinate system to the gantry frame, the object being scanned will rotate within the fixed frame (Fig. 13) and translate along the z-axis. Unlike the methods described by Rodríguez-Alvarez *et al.* where the system matrix for the cylindrical volume is made smaller by clever data recycling, the methods presented here don't attempt to explicitly make use of circular symmetries. Instead, the entire volume can be described as a set of simple trigonometric equations that virtually eliminate the need for any kind of system matrix storage. Future work will include the additional geometric formulation that describes how to eliminate the rotational dependency from the system matrix.

For simplicity, the geometry of the system will first be explained in two-dimensions only by focussing on the plane normal to the z-axis (the axis of rotation). The remainder of this thesis will only deal with the 2D geometry, so the formulation will henceforth be referred to as the *circular volume symmetry*. To make use of the circular symmetry of the system we choose to treat the plane as if it is made up of a series of concentric rings (Fig. 14), similar to the system CR-P created by Mora *et al.* [28]. Because the voxels in the plane have depth, from this point on the concept of a "plane" will be interchangeably referred to as a "disk".

By reading the scanner motor positions that are stored as private tags in the associated Digital Imaging and Communications in Medicine (DICOM) files, we are able to determine how far the object has moved along the z-axis and therefore perform image reconstruction with the correct disk within the volume. The properties associated with the z-axis will be discussed in section 4.4.

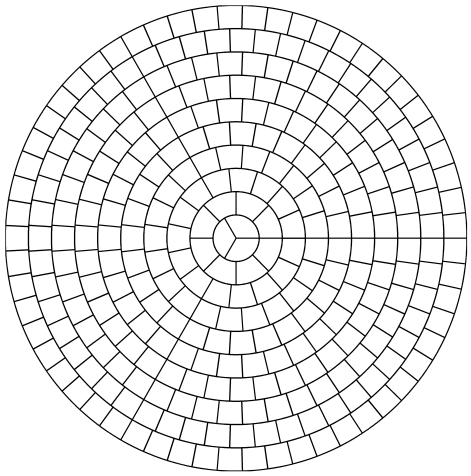


Figure 14: Visualisation of the first 10 rings in a disk.

Similar to CR-P method, we employ the same technique of keeping Δr constant for every consecutive ring, but our method differs in a number of ways. Instead of breaking up the disk into several identical segments, we choose to neglect this symmetric technique in favour of a simple formula that determines the number of voxels per ring. Let us define an arbitrary length a and attempt to create voxels that will be close to cubic with a volume a^3. Voxels will have area a^2 in the 2D plane and length a along the z-axis. Fig. 14 demonstrates how voxels become more square as distance from the centre increases, and Fig. 15 shows the geometry of a single 3D voxel.

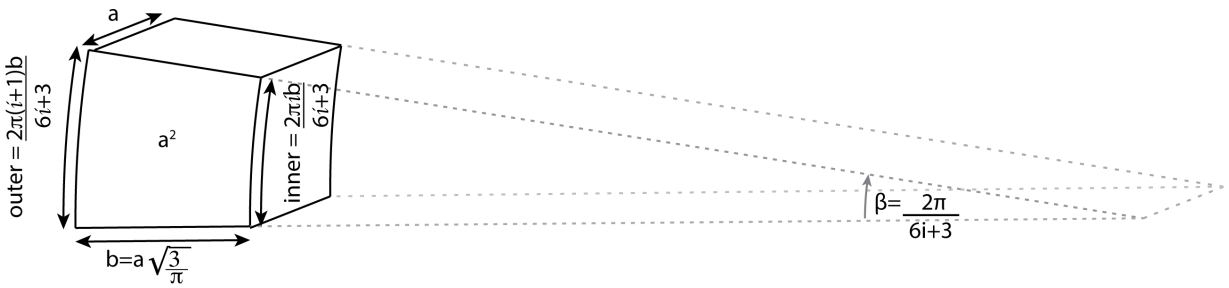


Figure 15: The almost cubic shape of a single voxel (MARS Bioimaging, 2014).

The gap or step size between successive rings is denoted by the variable b. Ring number is denoted by the variable i, with the first ring containing 3 voxels and is assigned the value $i = 0$. Each ring is made up of two circles, with the inner circle having radius $R_i^{(\text{inner})} = ib$ and the outer circle having radius $R_i^{(\text{outer})} = (i+1)b$.

A zero-based indexing system is used for the circular volume geometry in order to stay consistent with the way most programming languages index their vector or array elements. The voxel indexing system also starts at zero. For example, the central ring ($i = 0$) contains voxels {0, 1, 2}, and the second ring ($i = 1$) contains voxels {3, 4,..., 11}. Since the first ring ($i = 0$) is a circle with radius b containing three voxels with area a^2, we can relate the length a to the distance b with the relationship

$$\pi b^2 = 3a^2 \tag{4.3.1}$$

$$\Rightarrow \quad b = \sqrt{(3/\pi)}a \approx 0.977205a \tag{4.3.2}$$

The radial dimension is constant at $b = \sqrt{3/\pi}a$. Voxel faces become more and more square shaped as the radius of the associated ring increases. For example, the inner and outer arc lengths of ring 20 are 0.99837 and 1.04828 respectively.

The outer radius of ring i is $(i+1)b$, so by generalising (4.3.1) to work for any ring, the circular area of disk i is given by

$$\pi(i+1)^2b^2 = 3(i+1)^2a^2 \tag{4.3.3}$$

It will be noted that the right hand side of (4.3.3) is an integer multiple of a^2, which is convenient because we require an integer number of voxels in every ring. The area of ring i is calculated by finding the difference between the circular areas enclosed by consecutive rings,

$$3(i+1)^2a^2 - 3i^2a^2 = 3(2i+1)a^2 \tag{4.3.4}$$

From Equation 4.3.3 we note that there are a total of $3(i+1)^2 = 3i^2+6i+3$ voxels in an entire disk, and from Equation 4.3.4 we note that there are $3(2i+1) = 6i+3$ voxels in a single ring. Fig. 16 shows the quadratic relationship between the number of rings within a disk and the total number of voxels contained within the disk.

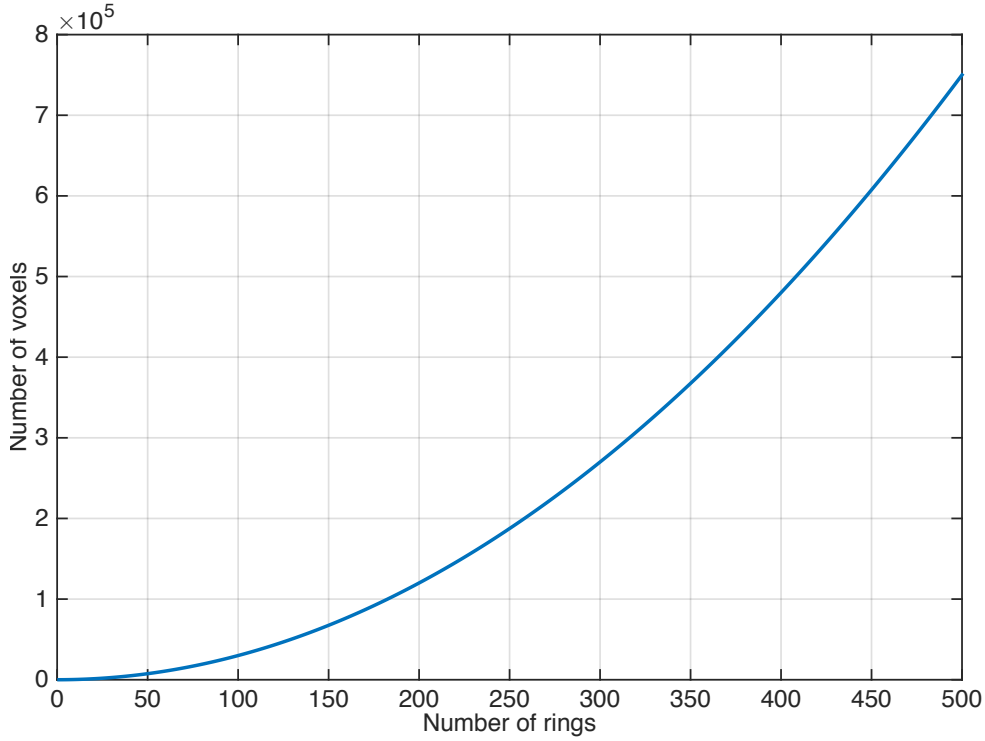


Figure 16: Quadratic relationship between the number of rings and the number of voxels in a disk.

For a disk with n rings, the radius R is given by

$$R_{\text{disk}} = nb \tag{4.3.5}$$

Along the z-axis there are k disks, making the total volume length $L = ka$. By making the substitution $i + 1 = n$ into Equation 4.3.3 we can calculate the total volume V as

$$V = 3n^2ka^3 \tag{4.3.6}$$

The voxels in ring i are separated by an angle δ_i, where

$$\delta_i = \frac{2\pi}{3(2i+1)} = \frac{2\pi}{6i+3} \tag{4.3.7}$$

The rotational angle β between the two edges of the j^{th} voxel in the i^{th} ring are calculated from the angles $\frac{2\pi}{3(2i+1)}j$ and $\frac{2\pi}{3(2i+1)}(j-1)$, measured in radians.

A quadrant system is employed throughout this thesis to help visualise some portions of the circular volume geometry. Features of the geometry often occur in pairs, but referring to them as "left" or "right" was confusing. For example, after the volume rotates, a point that was previously on the left and above the x-axis may now be on the right and below the $x - axis$. To solve this problem, the entire Cartesian plane was segmented into four quadrants: alpha, beta, gamma, and delta. The gamma quadrant resides in the $(+x, -y)$ region and the delta quadrant resides in the $(+x, +y)$ region. All code samples and graphical examples in this thesis occur above the x-axis, so only the gamma and delta quadrants are referenced. They are coloured red and green, respectively.

4.4 FAN-BEAM GEOMETRY

Fig. 17 is a simple example of four x-rays being emitted from the x-ray source, which proceed to pass through different regions of the reconstruction volume, and finally arrive at four separate pixels in the detector chip. An ART-based reconstruction algorithm requires knowledge of the distance travelled by each ray through every individual voxel. Fig. 18 shows the geometric properties of a single x-ray crossing a ring in the fan-beam geometry.

For every ring within the reconstruction volume, the calculation of path length requires the consideration of two cases:

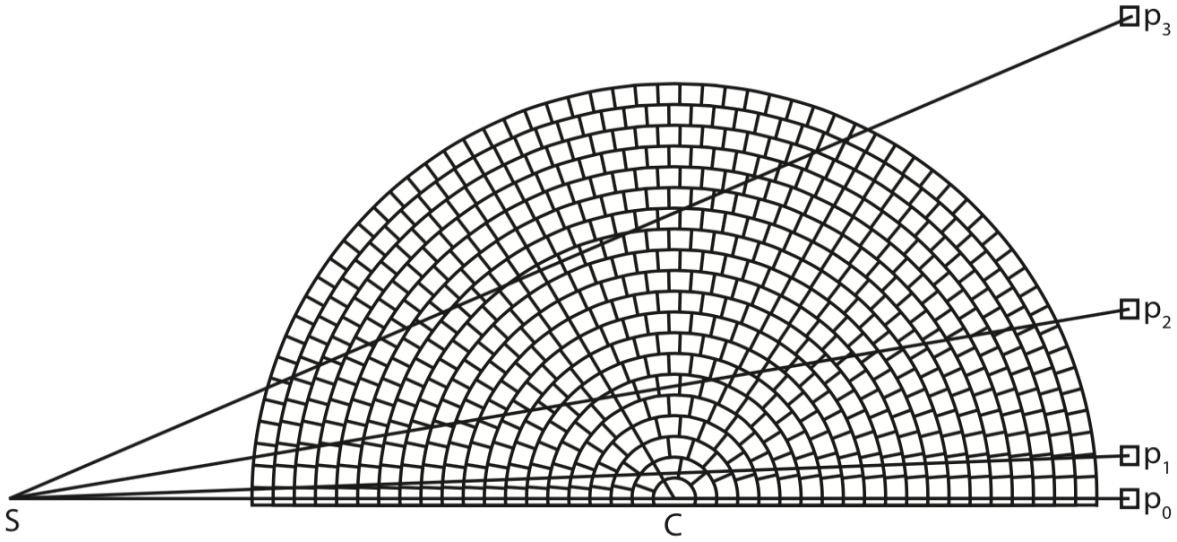


Figure 17: Four rays traverse the reconstruction volume and arrive at separate detector elements (MARS Bioimaging, 2014).

1. A ray line only cuts the outer radius of a ring, meaning that it is the inner-most ring to be intersected. This will only happen once per ray line, and the distance travelled will often be greater than for any other ring.

2. A ray line cuts both the inner and outer radii of a ring. This is the most common situation and occurs twice for every ring being traversed, except the inner-most ring. Fig. 18 demonstrates this case.

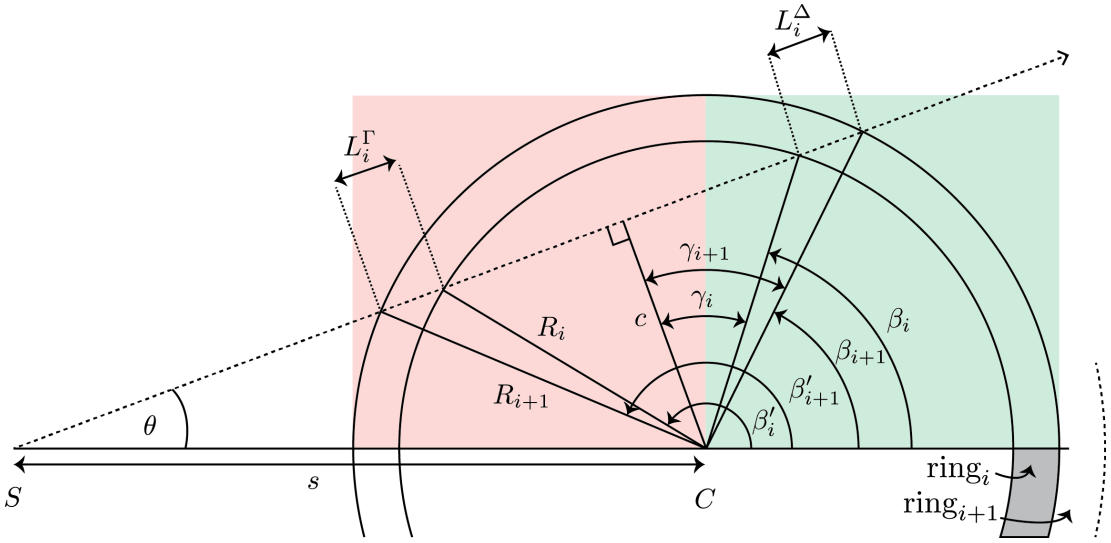


Figure 18: Path length through ring i can be calculated using simple trigonometry. The x-ray leaves the source, S, and intersects ring i twice. The path lengths, L_i^{Γ} and L_i^{Δ}, in neighbouring quadrants are identical (MARS Bioimaging, 2014).

The key to determining path length as shown in Fig. 18 relies upon the mathematical relationship between rings i and $i+1$, the angle of incidence θ, and the distance s from the x-ray source S to the centre of volume C. The angles and distances in Fig. 18 can be used to define the distance c as

$$c = s\sin\theta \tag{4.4.1}$$

$$= ib\cos\gamma_i \tag{4.4.2}$$

and thus

$$\gamma_i = \cos^{-1}\left(\frac{c}{ib}\right) \tag{4.4.3}$$

$$= \cos^{-1}\left(\frac{s}{ib}\sin\theta\right) \tag{4.4.4}$$

$$\text{where} \quad 0 \leq \gamma_i \leq \pi/2 \tag{4.4.5}$$

There are two path lengths to consider: one in the gamma quadrant, L_i^{Γ}; and one in the delta quadrant, L_i^{Δ}, of the circular volume. The laws of trigonometry dictate that these two lengths are equal, so the path length in the plane of the disk, $L_i = L_i^{\Gamma} = L_i^{\Delta}$ is given by

$$L_i = (i+1)b\sin\gamma_{i+1} - ib\sin\gamma_i \tag{4.4.6}$$

Extending the fan-beam geometry into the third-dimensional creates a cone-beam geometry, meaning that we also need to consider the x-ray path along the z-axis, as shown in Fig. 19.

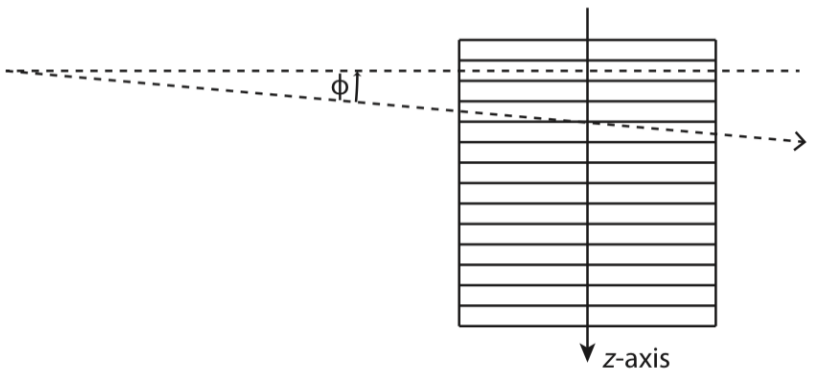


Figure 19: Dependence on ϕ further complicates path length calculations because the ray may pass through several neighbouring disks within the volume (MARS Bioimaging, 2014).

The ϕ dependent path length L_i^{ϕ} is found by including the factor $\frac{1}{\cos\phi}$ for all rays which don't pass through the central disk, as described in Equation 4.4.7.

$$L_i^{\phi} = ((i+1)b\sin\gamma_{i+1} - ib\sin\gamma_i) \,/\, \cos\phi \tag{4.4.7}$$

Because the ϕ component of the cylindrical volume geometry (Fig. 19) isn't a dependency in the two-dimensional circular geometry model, only the θ component will be investigated in this thesis.

A cylindrical volume used in a MARS-CT scan may consist of disks containing over 500 rings and therefore nearly 10^6 voxels. A simple way to reduce the number of calculations necessary is to first calculate the inner-most ring to be intersected by a given ray at angle θ (Algorithm 1) allowing the algorithm to completely ignore any rings that lie beneath it. By only operating on a subset of the disk data, the reconstruction process will be far more efficient. For a given ray we know the angle, θ, and the disk has constant properties such as the source-to-object distance (SOD), s; radius, R_d; and number of rings, n (Fig. 19). Because the line $c = s\sin\theta$ lies orthogonal to the ray line, the connecting point on the line is also the nearest point to the centre of the disk, C.

Algorithm 1: Finding the inner-most ring intersected by the ray line

```
/* findInnerMostRing function                                          */
```

input : $\theta, numberOfRings, diskRadius, SOD$

output : $innerMostRing$

$c \leftarrow SOD \times \sin(\theta)$

$ringThickness \leftarrow \dfrac{diskRadius}{numberOfRings}$

```
/* taking the floor of the ratio maintains the zero-based indexing system  */
```

$innerMostRing \leftarrow \left\lfloor \dfrac{c}{ringThickness} \right\rfloor$

Each ring is essentially a circle with a smaller circle cut out of its centre, and we can use simple Euclidean geometry techniques to calculate the x-ray path length through an individual ring [30]. This technique is necessary when it comes to finding the path length through the inner-most ring that was intersected, due to the fact that Fig. 18 only applies when the ray has two entry and exit points. The Euclidean geometry method can also be used as a complete replacement for the method described earlier in this section, but it would require calculating the length through circle

i and then subtracting the length through circle $i-1$. This idea is explored more in Section 4.5. Because the path length through each ring depends directly on the ring directly inside of it, this method could not be easily designed as a parallelised algorithm.

Finding the intersection point(s) of the line $ax+by=c$ and the circle $x^2+y^2=r^2$ we can use the two equations [30]

$$x_{1/2} = \frac{ac \pm b\sqrt{q}}{a^2+b^2} \tag{4.4.8}$$

$$y_{1/2} = \frac{bc \mp a\sqrt{q}}{a^2+b^2} \tag{4.4.9}$$

$$\text{where} \qquad q = r^2\left(a^2+b^2\right) - c^2$$

For the term q there are three cases that need to be considered:

- $q > 0$: there are two intersection points, meaning that the x-ray must pass in one side of the ring and out the other side. This ring may or may not be the most central ring to be intersected.
- $q = 0$: there is only one intersection point and the x-ray must simply graze the edge of a ring. In this case the ring will likely be ignored.
- $q < 0$: there is no intersection.

Equations (4.4.8) and (4.4.9) form the primary component of Alg. 5 presented in Chapter 5 and is thoroughly tested with the Circular Volume Simulator in Section 5.3.

4.5 PARALLEL-BEAM GEOMETRY

Most reconstruction algorithms described in literature or available as code samples online employ a parallel-beam x-ray source model. This choice allows the discussion to focus on reconstruction methods without being distracted by source beam geometries, and when most researchers use the same beam geometry, their experimental results can be compared and discussed in a more consistent way. An algorithm to determine circular volume voxel intersections for a parallel-beam x-ray source was developed in MATLAB and then implemented using MEX and CUDA in Chapter 5.

Presented in this section is an alternate method of calculating the path lengths within a ring. Because the parallel-beam geometry has no reliance on θ (Fig. 20), the calculations are simpler than those described in Section 4.4. In terms of the path lengths L_i^{Γ} and L_i^{Δ}, both models share the same trigonometric properties. The parallel-beam geometry adds a new variable, h, which represents the elevation or height of the incoming x-ray. The value of h makes the task of finding the appropriate detector pixel a trivial matter.

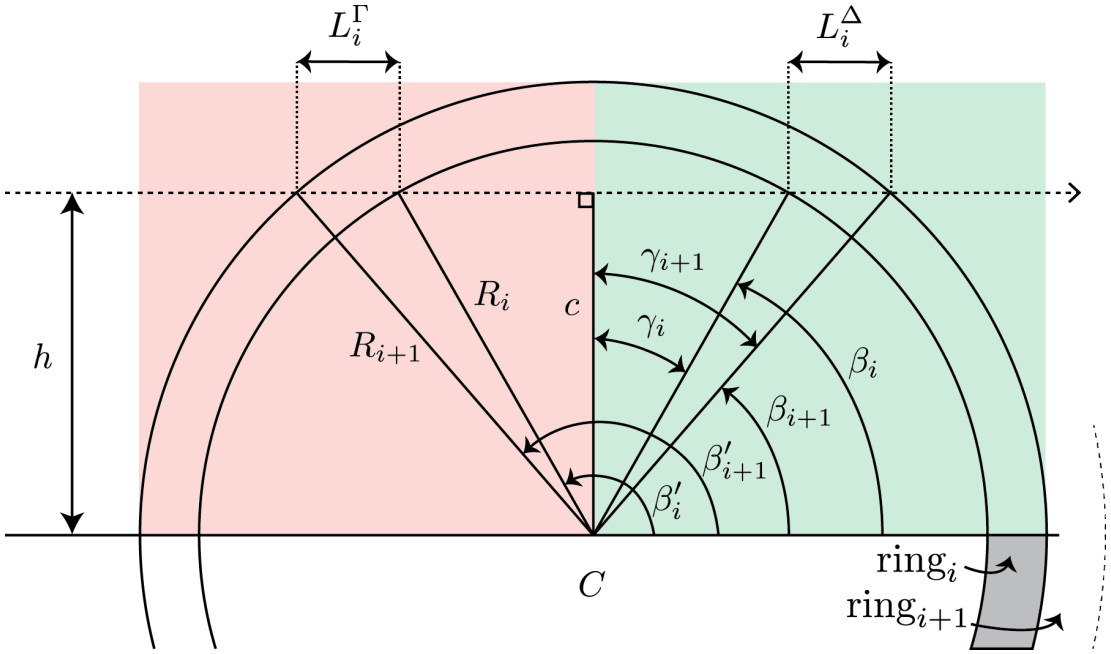


Figure 20: The circular geometry is intersected by a ray parallel to the x-axis (MARS Bioimaging, 2014).

The symmetry along the y-axis in the parallel-beam geometry means finding the path length across each ring in each quadrant is simply a matter of calculating the difference between the chord length of rings i and $i-1$. Combining variables from Figs. 20 and 21 we obtain

$$L_i = L_i^{\Gamma} = L_i^{\Delta} = \frac{a_i - a_{i-1}}{2} \tag{4.5.1}$$

$$= \sqrt{h}\left(\sqrt{2R_i - h} - \sqrt{2R_{i-1} - h}\right) \tag{4.5.2}$$

where R_i and R_{i-1} are the index numbers of rings i and $i-1$ multiplied by the constant ring thickness.

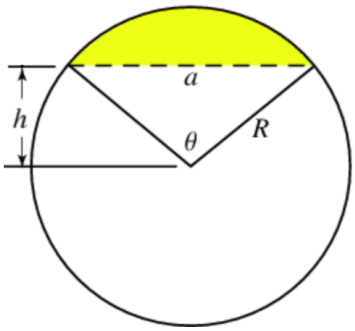


Figure 21: The chord length is determined by the formula $a = 2\sqrt{h(2R-h)}$ [4].

4.6 ALLOCATION OF RING PATH LENGTH TO VOXELS

One approach to speeding up image reconstruction is to take advantage of the fact that neighbouring voxels are very likely to contain the same materials. This assumption allows us to save time by neglecting to calculate the exact path length through every individual voxel. Instead, we can calculate the path length through every individual ring and use simple functions to calculate which voxels must have been intersected. Finally, the length through the ring is evenly divided amongst the specified voxels. If a voxel is chosen to be included in the distribution of the ring path length, the ray line within the voxel will have a length that is comparable to the voxel dimensions (Fig. 22).

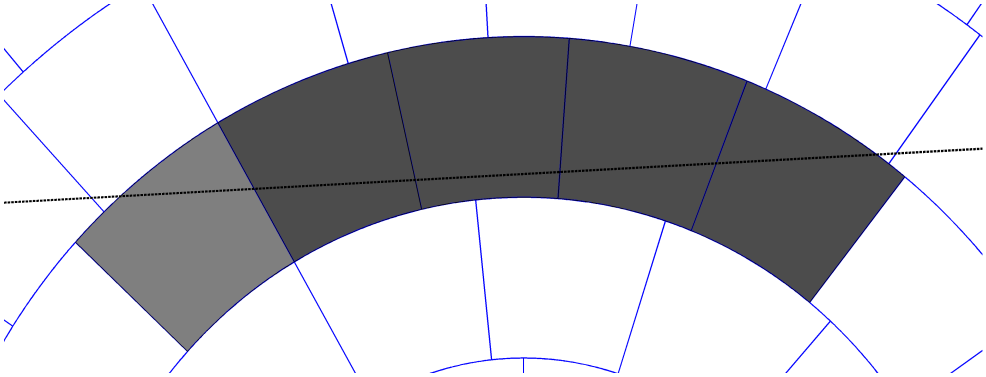


Figure 22: Five voxels intersected by a ray line at an angle of 3°. The left-most (light grey) voxel will likely be ignored by the reconstruction algorithm.

Because every ring has an identical thickness, we can use this knowledge to estimate what the average path length across a voxel might be. If we were to assume that the ray line passes through the centre of the voxel (Fig. 23), we can infer that the length is likely to be similar to the arc length that corresponds to the radius halfway between the inner and outer radii of the ring.

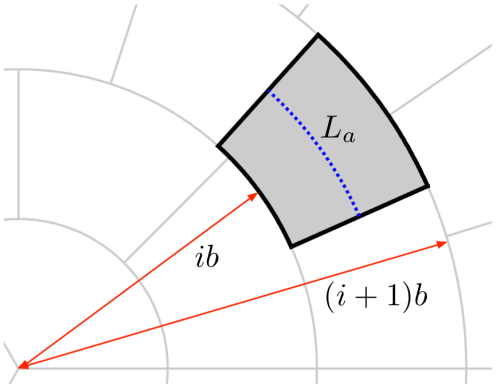


Figure 23: Calculation of $L_a = \frac{\pi}{3}b$.

The radius R is calculated from the two ring radii, R_i and R_{i+1}, and is given by

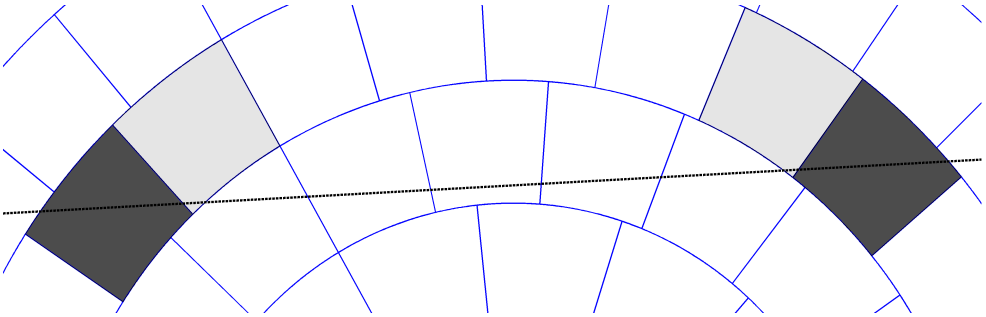
$$R = \frac{R_i + R_{i+1}}{2} \tag{4.6.1}$$
$$= \frac{ib + (i+1)b}{2} \tag{4.6.2}$$
$$= \frac{2i+1}{2}b \tag{4.6.3}$$

The arc length L_a through a voxel of ring i is given by

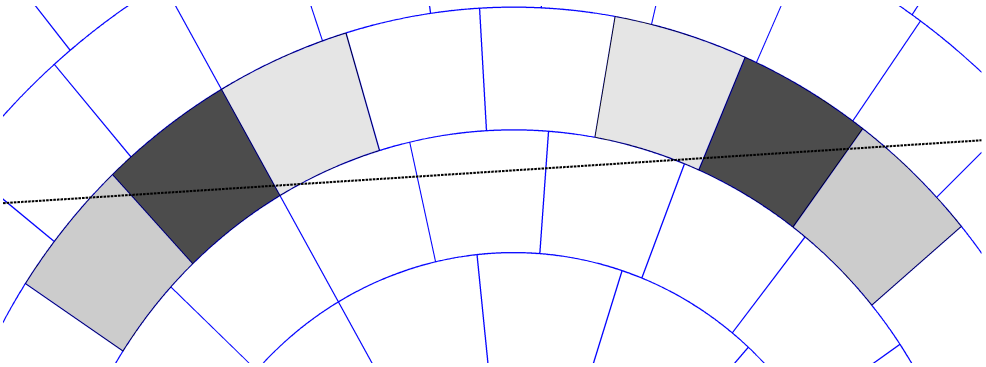
$$L_a(i) = \frac{2\pi R}{\text{number of voxels}} \tag{4.6.4}$$
$$= \left(\frac{2\pi}{3(2i+1)}\right)\left(\frac{2i+1}{2}\right)b \tag{4.6.5}$$
$$L_a = \frac{\pi}{3}b \tag{4.6.6}$$

where b is the constant ring thickness, and for n rings the ring index ranges from $0 \ldots n-1$. With no dependence on i, L is a constant and only needs to be calculated once. The calculation of L_a is only necessary when the ring thickness changes, which will usually only happen if the reconstruction algorithm decides to increase or decrease the number of rings in a volume while keeping the same volume radius.

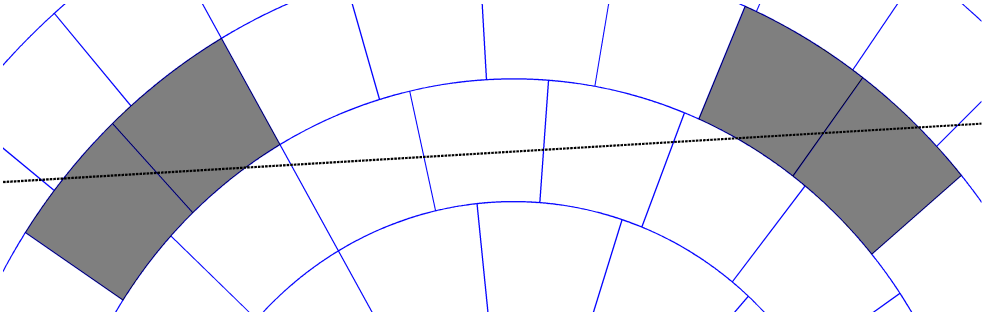
After the total path length inside the ring, L_r, and the general path length through a voxel, L_a, have been calculated, the algorithm will divide L_r by L_a and round the result to the nearest integer. This gives a good approximation of how many voxels in the ring have had a significant intersection, N. The next step is to decide which of the intersected voxels will be included in the reconstruction step, considering that an actual path length through each voxel is never calculated. Fig. 24 shows three typical scenarios for this problem.



(a) Four voxels intersected by an x-ray at an angle of 3°, but only the darker two will be included in path length calculations.



(b) Six voxels intersected by an x-ray at an angle of 3.5°, but only the darker two will be included in the path length calculation.



(c) Four voxels intersected by an x-ray at an angle of 3.25° and the ray line appears to intersect each equally. A decision must be made over which two will be allocated a path length.

Figure 24: Minor variations in x-ray angle have profound effects over which voxels will be included in the path length calculations.

The procedure occurs in several steps, and begins by finding the voxels that sit at the extremities of the ray line within the ring. The voxels in-between them are determined, giving a complete list of candidate voxels, N_c. The procedure then continually subtracts voxels from N_c until the size of N_c equals N. Fig. 24a shows the simplest scenario where two voxels are intersected in each quadrant, but only one on each side will ultimately be selected for inclusion in N_c. Fig. 24b shows a scenario where intersected voxels sit on either side of the voxels we ultimately want to keep in each quadrant. Also, there are two voxels in the middle which have no interaction at all,

but are still initially included in N_c before their removal. Fig. 24c shows a scenario where it's not clear which voxel in each quadrant should be included in N_c. The ray line passes through each voxel almost equally, and yet the path length algorithm will claim that only two voxels must be included in N_c and the other two must be ignored. These scenarios demonstrate how voxel selection is a delicate matter and must be dealt with in a consistent way in order to minimise image artefacts.

In all rings except the inner-most ring, the intersected voxels lie in two separate groups within the gamma and delta quadrants of the volume. The path length algorithm iteratively switches between quadrants, removing one voxel at a time from the list N_c until the termination condition is met. This procedure is implemented as the *voxel intersection algorithm* in chapter 5.

4.7 INDEX CONVERSION

Voxels in a cylindrical volume are numbered starting in the central ring and increasing outward in a spiral-like manner. There is no one-to-one correlation between the voxel positions in the circular geometry and a traditional square grid that is associated with storing values in an array. Given the huge number of voxels with a disk, finding the linear index of the corresponding vector element needs to be extremely fast. Two conversion utilities (Algs. 2 and 3) were developed to convert between circular geometry $(disk, ring, voxel)$ indices and linear indices.

In order to find the voxel elements in neighbouring disks, it is simply a matter of adding an offset equal to the number of voxels in an entire disk. This procedure is trivial and is not included in Algs. 2 and 3.

Algorithm 2: Convert circular geometry index to linear index.

```
/* ConvertToLinear                                              */
/* Inputs are zero-based indices                                */
input  : voxelNum, ringNum
output : index
/* The equation isn't valid for the central ring.               */
if ringNum > 0 then
    ringIndex ← 3 × (ringNum)^2 + 6 × ringNum + 3
else
    ringIndex ← 0
end
index ← ringIndex + voxelNum
```

Algorithm 3: Convert linear index to circular geometry index.

```
/* ConvertToCircular                                                    */
```
input : $index$

output: $voxelNum, ringNum$

```
/* Calculate ring number using the positive integer result from the
   quadratic formula                                                    */
```
$a \leftarrow 3$

$b \leftarrow 6$

$c \leftarrow 3 - index$

$$ringNum \leftarrow \left\lceil \frac{-b + \sqrt{b^2 - 4ac}}{2a} \right\rceil$$

```
/* Calculate the voxel number by subtracting the total number of voxels
   found in all smaller rings                                           */
```
$$voxelNum \leftarrow index - 3 \times (ringNum)^2$$

4.8 CONCLUSION

The circular volume geometry and associated voxel intersection algorithm will significantly reduce the amount of system matrix storage required. This is due to the way that x-ray path length will be calculated on a ring-by-ring basis, with the resulting path length evenly distributed to only the most heavily intersected voxels. Compared to the algorithm presented by Rodríguez-Alvarez *et al.* (2013) which calculated weighting factors based on intersection areas within each voxel, the methods described in this thesis are significantly faster. The formulation of the cylindrical volume geometry model presented in this chapter makes a significant contribution to the new ART-based reconstruction algorithm currently being developed for use in MARS scanners.

4.9 SUMMARY

- This chapter has presented part of the formulation of a novel approach to cylindrical volume geometry developed by Prof. Phil Butler and Dr. Peter Renaud (University of Canterbury). This model will allow a significant reduction in the size of pre-calculated system matrices and represents a major contribution to the development of a new ART-based reconstruction algorithm for use in MARS scanners.

- Fan and parallel x-ray beam geometries were described, with the latter being used in the chapters that follow.
- The motivation and methods of the path length allocation method were explained.
- Two algorithms were presented that convert between the circular volume index system and the linear index system.
- Key cylindrical volume geometry concepts:
  - The volume is sliced into disks along the axis of rotation, and each disk is made up of rings.
  - Ring number is denoted by i, with the first ring being numbered $i = 0$.
  - Every ring contains $6i + 3$ voxels.
  - Every disk contains $3i^2 + 6i + 3$ voxels.
  - Every voxel has volume a^3.
  - The radial distance between rings is denoted b, giving each ring an inner radius of ib and an outer radius of $(i + 1)b$.
  - Indexing of voxels, rings, and disks starts at zero.

5

SYSTEM MATRIX CREATION

This chapter explains the motivation and methods behind designing an efficient algorithm to compute voxel path lengths in the circular volume geometry. The algorithm was prototyped in MATLAB because it allowed for fast code development and easy debugging. A graphical tool was developed that allowed a user to simulate and manipulate an accurate representation of the circular volume, and it used two methods to calculate intersected voxels. The first relied entirely on MATLAB functions such as `polyxpoly`, and the second was an algorithm developed based on the geometry formulation introduced in chapter 4. The simulator tool allowed both models to be compared, thus ensuring the accuracy of the geometry formulation equations. In order to establish the appropriate high-performance hardware that the algorithm should be designed for, the code was ported into C++ and CUDA and executed within MATLAB through the MEX interface. Comparing the performance of all implementations showed that the algorithm would be most suited to run as a CPU-based parallel program.

Sections 5.1 and 5.2 discuss the background and related research to the voxel intersection algorithm developed in this chapter; section 5.3 describes the challenges involved with creating the algorithm within MATLAB; section 5.4 gives background to the CUDA parallel computing platform; section 5.5 discusses implementing the algorithm as C++ and CUDA subroutines; section 5.6 explains the methodology and results of speed comparisons between the different implementations; and the chapter ends with a discussion and a conclusion in sections 5.7 and 5.8, followed by a summary in section 5.9.

5.1 INTRODUCTION

As with any voxel-based CT image reconstruction algorithm, one of the most important and most time-consuming parts is the calculation of which voxels were traversed or intersected by a given photon. Depending on factors such as the x-ray tube current, these photons can count in the tens of thousands. For algorithms based on ART, having a list of intersected voxels isn't the end of the story. Due to the existence of the distance variable s in the Beer-Lambert Law (Eq. 2.3.1), the

unique path length that the photon travelled within each individual voxel is also required as part of the reconstruction process.

Implementation of the circular volume geometry is intended to reduce the number of photon path length calculations by several orders of magnitude, thereby improving the speed of reconstruction and providing a more commercially viable MARS scanner to the end user. A list of voxels intersected and the associated path lengths are calculated for every projection angle and the results stored in a look-up table that can be reused for every reconstruction. These look-up tables can be converted to represent the system matrices discussed in earlier chapters. Even though the look-up tables were created successfully, their sizes ranged from several hundred megabytes to multiple gigabytes. In the future when the algorithm is updated to consider cone-beam x-ray sources, the look-up tables will likely become significantly larger. For this reason, it was important to consider the possibility that voxel intersection calculations might need to be calculated on-the-fly during the reconstruction process, and henceforth the performance of each implementation was assessed in section 5.6. Chapter 6 follows on from this development by focusing on methods for drawing the circular volume geometry to the screen, and the degree to which the geometry can be integrated into a FBP algorithm.

The software development described in this chapter lays an important foundation for what will become a major part of the new reconstruction method being developed by the MARS group. The circular volume geometry will provide a significant performance boost to reconstructions performed on future MARS scanners.

5.2 RELATED WORKS

Implementations of parallelised reconstruction algorithms that underly medical imaging modalities such as CT, MRI and PET have been researched extensively. With the availability of increasingly powerful computer hardware such as multi-core CPUs and GPUs, methods that were once thought too difficult or too computationally intensive are now becoming mainstream. This section presents some of the more interesting research conducted in recent years.

Image reconstruction by iterative methods involves solving a linear system, eg. $Ax = p$, where A is a system matrix simulating CT functioning and depends upon the projection number and projection angle, x is a column vector representing image intensities, and p is a column vector containing the collected projection data [12]. The CUDA CUSPARSE library allows basic linear

algebra operations to be performed on sparse matrices and vectors such as A and x. Flores *et al.* (2014) ran iterative reconstruction benchmarking tests on a range of matrix sizes using a GPU computing node made up of 2×Intel Xeon X5660 (each with 6 cores) and 2×NVIDIA Tesla M2050 GPUs (each with 488 cores). They found that while the computation time using the CPU increased dramatically with matrix size, the computation time using the CUSPARSE libraries on the GPU remained relatively constant. As the matrices got larger, the speed benefits of using the GPU became increasingly prominent. For example, reconstruction time of the largest system matrices took 24.4s on the CPU and 5.3s on the GPU [12]. Similarly, Xin *et al.* (2013) implemented SART using CUDA for cone-beam CT (CBCT) reconstruction algorithms and benchmarked the resulting speedups. Improvements varied extensively, depending on the GPU used and the size of the sparse matrix tested, which ranged from 16x16 up to 256x256 elements. Of the four GPUs tested, the NVIDIA GTX 480 showed the largest speedup of 61.88x when dealing with a 128x128 matrix [31].

One of the drawbacks of using CBCT for image guided radiation therapy (IGRT) is the associated large radiation dose that the patient will receive. A common technique is to lower the radiation dose and thus try and reconstruct CBCT images from the relatively noisy and under-sampled projection data that results. Jia *et al.* (2010) aimed to develop fast GPU-based algorithms to perform this type of reconstruction. When reducing the number of x-ray projections and/or the mAs level, the data collected cannot be used with conventional FBP algorithms because the images reconstructed from the highly under-sampled and/or noisy projection data are found to be of too low a quality to be clinically useful. The total variation (TV) method potentially allows for the recovery of signals from incomplete measurements through various optimisations, however the computation is time-consuming due to an absence of efficient algorithms that can deal with the huge data sets encountered. The GPU-based algorithms developed by the authors were found to be about 100 times faster than non-TV based approaches, as well as delivering an estimated 36–72 times dose reduction to the patient [32]. As the size requirements of CT acquisition data gets larger, modern GPUs will struggle to provide enough memory to store all the data needed for the reconstruction process.

One method for optimising memory usage and computation time involves empty space skipping and performing multi-resolution reconstruction. Zhao *et al.* (2013) implemented these ideas by first subdividing the volume into equally sized blocks. They then performed an initial low-resolution reconstruction on each block to identify empty blocks, which are then be packed into a new volume made as small as possible. The remaining blocks are rearranged to more efficiently

generate the high-resolution volume. When a volume contains a large amount of air space, the reduced volume now fits into the texture memory on the GPU and requires no slow data swaps during reconstruction. Also, the non-empty blocks are the only ones used during the iterative reconstruction, which improves the computation time significantly [33].

Multi-Thread Scheduling (MTS) is another computational technique that was implemented by Zhu *et al.* (2012). It allowed 3D CT image reconstruction to take place using the FDK algorithm while efficiently managing the computation that was being performed by the GPUs, as well as the data reading and writing being performed on hard disks. The authors introduced a concept known as Complete Reconstruction (CR) that encompasses the entire reconstruction process, from reading the data on disk, through to image reconstruction, and finally writing the images to disk. They found that calculation time could be reduced significantly by implementing MTS [34].

5.3 MATLAB IMPLEMENTATION

MATLAB is an excellent prototyping tool, and all aspects of algorithm development were heavily influenced by being able to use MATLAB to draw a realistic representation of the circular volume geometry to the screen. It was crucial to be able to overlay this figure with that of an x-ray traversing the volume. By programming MATLAB to determine which individual voxels were intersected by the x-ray line it was possible to simultaneously develop a efficient algorithm to perform the same function but without having to rely on any of MATLAB's functions and toolboxes.

The voxel intersection model is made up of a line representing the x-ray path and an object representing the circular volume geometry containing r rings and v voxels, where each voxel is an individually drawn polygon. Both the fan-beam and parallel-beam models are discussed in this chapter.

The model is laid out on a cartesian plane with the circular volume centred at the origin. The x-ray source, S, is positioned in the $-x$ direction and the detector array is positioned in the $+x$ direction, which is in turn made up of several detector elements stretching in the $+y$ and $-y$ directions. An x-ray path is modelled as a straight line connecting S to the detector pixel intersected by the photon, D. The distance from the x-ray source along the x-axis to the front of the detector is the SDD and from the centre of the circular volume to the detector is the SOD.

When it came to determining which voxels were intersected by a particular ray, MATLAB was used to develop a visual representation of a small sample of rings. Visualising hundreds of rings

at once is not feasible, nor should it be desirable, since any algorithm designed for a small set of rings can be easily scaled up to a volume of any size. The non-symmetrical voxel layout in the circular volume geometry provided a challenge when it came to algorithm design. Unlike the models developed by Rodríguez-Alvarez *et al.* and Jian *et al.*, there were no repeating segments in this circular geometry, and every ring can be considered to be a separate entity. There is no obvious pattern that can predict which voxels would be intersected for specific volume parameters. Designing an algorithm that accurately performs this task, while not difficult, took time and patience to complete to the required level of accuracy. Being a highly visual problem and for the purpose of algorithm verification, a program called the Circular Volume Simulator was built using MATLAB (Fig. 25). Only the parallel-beam geometry is displayed in the following figures, but the fan-beam geometry is entirely similar.

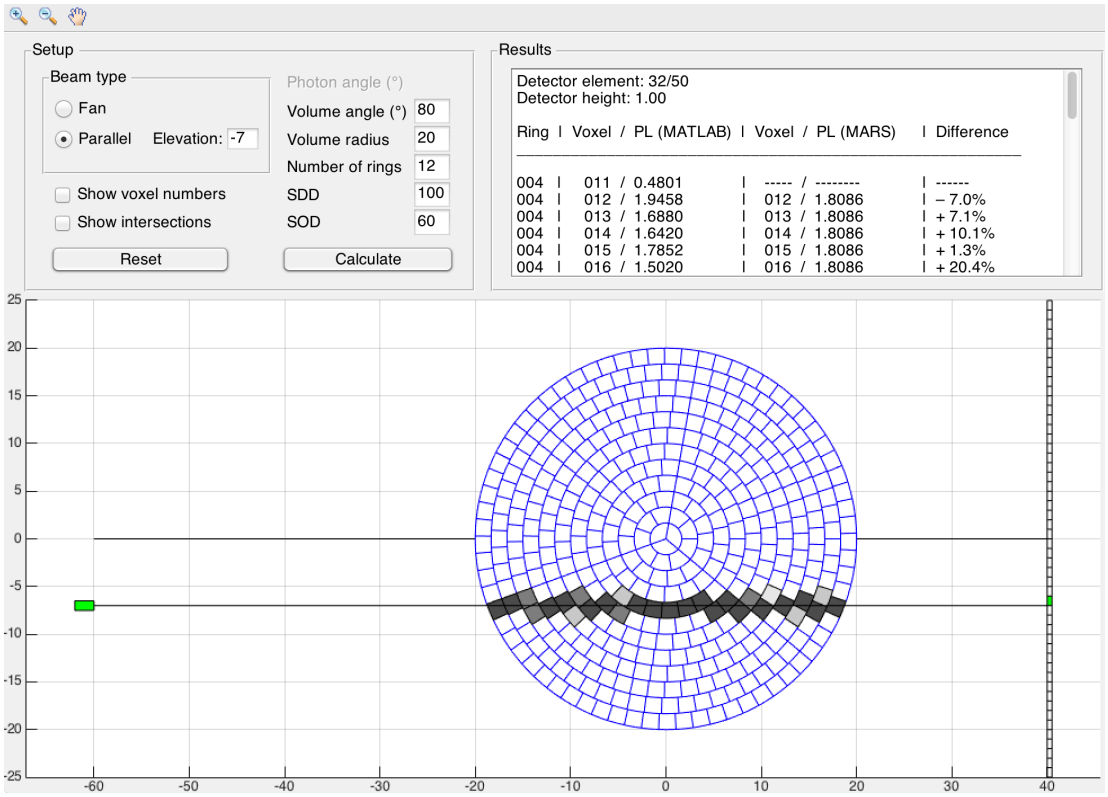


Figure 25: Fan beam model configuration displaying x-ray source, x-ray path, circular volume, and detector elements. Intersected voxels are coloured with varying shades of grey, where a darker shade indicated a longer intersection. The intersected detector element is coloured green.

A simplified representation of the scanner was drawn to the screen allowing the user to see a single x-ray leaving the source, travelling through the circular volume, and finally intersecting a detector element. The interactive graphical user interface (GUI) allowed the user to adjust several parameters and update the plot to see how changes will effect the final result (Fig. 26).

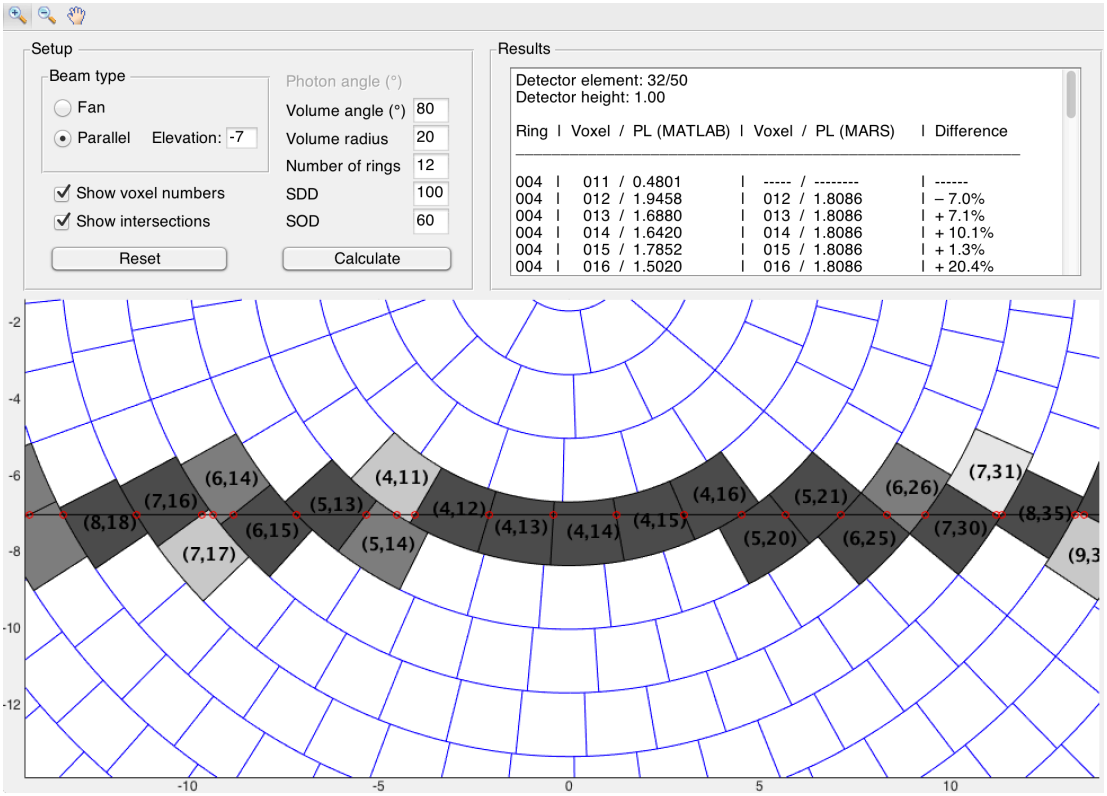


Figure 26: Close up view of the circular volume using the same parameters as Fig. 25.

The simulator contained many editable parameters, including: x-ray angle, θ (fan-beam); x-ray elevation, h (parallel-beam); volume angle, α; volume radius; number of rings; SDD; and SOD. While the relative dimensions and distances of the objects are all to scale, the lengths themselves are unit-less. The user has the option to show or hide voxel numbering labels, and the toolbar has standard zoom and pan tools for manual graph adjustment.

Every voxel drawn to the screen is an individual polygon created by a subroutine of the program, and to maintain smooth curves every arc consists of 100 data points. Arcs containing more points were trialled, but this didn't improve the quality of the image or the accuracy of the calculations, and instead resulted in a crippling load on system memory. Attempting to draw a volume with more than 25 rings also caused a severe slowdown in system responsiveness. The simulator was programmed to display an error message if the user attempted to draw too many rings. The other variables were also bounds checked for consistency.

The Circular Volume Simulator calculated and compared the path length calculation algorithm developed in this thesis to an algorithm that employed only MATLAB functions. The simulator incorporates several useful features to help the user check for parity between the two methods. The level of voxel shading is determined by the distance travelled by the x-ray within the voxel,

providing a simple visual indication of the level of x-ray interaction. By adjusting parameters by small increments, the user can observe how the changes effect the degree of x-ray intersection within each voxel.

The simulator prints the ring and voxel index in the centre of any intersected voxels, saving the user from having to manually count around the ring to check algorithm consistency (Fig. 26). For n rings, the ring index, $i = 0 \ldots n-1$, and the voxel index, $v = 0 \ldots (6i+3)-1$. To ensure that the algorithms were operating as expected, it's important to know that the voxel being intersected in the image matches that which has been calculated by the theoretical equations.

Algorithm 4: VoxelMATLABFun - Calculate fan-beam voxel intersections and path lengths with MATLAB's `polyxpoly` function, and draw the ray line and volume to the screen.

```
/* Voxel / PL (MATLAB) function                                          */
input  : photonAngle, volumeAngle, numRings, SOD, SDD
output : pathLengthArray, detector
/* calculate ray line by placing the object/volume at the origin        */
startPoint ← (−SOD, 0)
endPoint ← (SDD − SOD, SDD × tan(photonAngle))
rayLine ← [startPoint, endPoint]
for i ← 1 to numRings do
    /* calculate number of voxels in the ring                           */
    numVoxels ← 6 × (i − 1) + 3
    for j ← 1 to numVoxels do
        /* calculate voxel polygon points                               */
        [voxelPolygon] ← createVoxel(j)
        /* calculate ray line intersection                              */
        [points] ← polyxpoly(rayLine, voxelPolygon)
        if points ≠ nil then
            pathLength ← calculatePathLength(points)
            pathLengthArray[i, j] ← pathLength
        end
        drawVoxel(voxelPolygon)
    end
end
drawRayLine(rayLine)
```

When the user clicks the "Calculate" button, both path length algorithms are executed and the results displayed in two columns of a table. The first algorithm (Alg. 4) was called Voxel-MATLABFun and also draws the volume to the screen. The points that make up each voxel are stored in an array. This method relies entirely on MATLAB and the function `polyxpoly` which returns a list of points (marked on the graph as red circles) that occur at the edge of the polygon where the x-ray line and the voxel polygon meet. The function `polyxpoly` also determines which

rectangular detector element was intersected by the ray. VoxelMATLABFun iterates over every polygon in the volume and the path length of every intersection is derived from the pair of (x, y) intersection points using simple trigonometry. The indices of the intersected ring and voxel are recorded along with the precise path lengths through the voxel, and the results are displayed in the "Voxel / PL (MATLAB)" column of the simulator table. This method can be considered brute force because every voxel in the volume is tested and no attempt at optimisation is made. Instead of testing every single voxel for an intersection, it should be possible to create a smart algorithm that "follows" the ray line and only considers voxels that lie in its immediate vicinity.

Algorithm 5: VoxelMARSFun - Calculate voxel intersection points and path lengths with the fan-beam geometry.

```
/* Voxel / PL (MARS) function                                          */
input  : θ, numberOfRings, diskRadius, SOD
output : pathLengthVector
/* find the inner-most ring to be intersected with Algorithm 1         */
innerMostRing ← findInnerMostRing(...)
/* check for intersection points for the inner-most ring using (4.5.1) and
   calculate the path length                                           */
q ← r^2 (a^2 + b^2) − c^2
if q > 0 then
    /* two points of intersection, so (4.4.8) and (4.4.9) are applied  */
    (x1, y1) ← ( (ac + b√q)/(a^2 + b^2), (bc − a√q)/(a^2 + b^2) )
    (x2, y2) ← ( (ac − b√q)/(a^2 + b^2), (bc + a√q)/(a^2 + b^2) )
    pathLength ← √((y2 − y1)^2 + (x2 − x1)^2)
    pathLengthVector[innerMostRing] ← pathLength
else
    /* no path length allocation                                       */
end
/* calculate path length for all remaining rings with (4.4.4) and (4.4.6) */
for i = innerMostRing + 1 to numRings do
    c ← SOD × sin(θ)
    γ_{i+1} ← cos^{-1}( c / outerRadius[i] )
    γ_i ← cos^{-1}( c / innerRadius[i] )
    pathLength ← outerRadius[i] × sin(γ_{i+1}) − innerRadius[i] × sin(γ_i)
    /* every ring has two identical ring crossings                     */
    pathLengthVector[i] ← 2 × pathLength
end
```

The second algorithm (Alg. 5) was called VoxelMARSFun and attempted to generate the same information as the first, but instead makes use of the formulations derived in chapter 4. The algorithm begins by finding the inner-most ring so that the more centralised rings can be ignored. Unlike VoxelMATLABFun which calculates the path length for every individual voxel, VoxelMARSFun calculates the path length through each ring using trigonometry and the quadratic formula, storing the results in an array. In a separate but trivial algorithm (not listed), the indices of the intersected voxels are determined by using the volume rotation angle as an offset and again using trigonometry. The calculated path length is then divided evenly between the selected voxels. The remaining columns of the simulator table relate to these results, where the "Voxel / PL (MARS)" column displays the voxel index and path length calculated by the model. The percentage difference between the precise path length calculated by MATLAB and the approximate path length calculated by the model is printed in the final column, giving the user an idea of how the two models compare to one another. Because the model ignores voxels with minimal intersections, some rows in the table display blank lines. For the purposes of performance testing, VoxelMARSFun was run as a parallel process by using a `parfor`-loop.

5.4 CUDA

There is a great deal of complexity that underlies GPU programming, but two of the most important aspects relevant to this thesis, thread handling and memory management, are briefly introduced in this section.

GPUs consist of a number of steaming multi-processors (SMs) which are attached to eight or more stream processors (SPs). On NVIDIA GPUs, SPs are known as CUDA cores and operate up to 32 parallel sets of instructions. A typical GPU may consist of up to 30 SMs, each of which may contain 8 SPs. From a hardware perspective, GPU performance is largely determined by the number of SPs present and the available bandwidth to global memory. At the centre of the CUDA model is the idea of the thread, and the model is built up from there into groups called warps, blocks, and grids [15].

The *thread* is the most basic component of a parallel program, and exists in both GPU programming and multi-core CPU programming paradigms. A single thread is essentially a small serial program, and a GPU is capable of running thousands of threads concurrently. The capabilities of CPUs and GPUs are often compared at the thread level, where CPUs are designed to run a small

number of complex threads, and GPUs are designed to run a large number of simple threads. In CUDA, a function that will be run in parallel is called a *kernel*, and the GPU will attempt to launch a kernel as tens of thousands of concurrent threads. Each thread will operate on its own small piece of the greater problem.

Threads are grouped into sets of 32 called a *warp*, and warps are initiated as a whole. The next warp in the queue is not able to begin until the previous warp finishes, which highlights the importance of making sure that every thread in a warp is given a similar amount of work to do. If one thread takes a lot longer to finish than the rest, a large amount of processing power is wasted. The ideal case for a single warp is to have a single fetch from memory which is then broadcast to all threads within that warp, but this is usually possible in practice. The size of a warp may depend on several factors, but perhaps the most important one is to do with the extent of branching in the kernel. A warp is a single unit of execution, and branching causes a divergence in the execution flow. On a GPU, each branch must be executed in sequence, before all branches are resolved and the threads converge. A kernel that contains a large degree of branching will result in poor computational performance.

When it comes to kernel invocation, *blocks* are how threads are organised. When a kernel is launched, the programmer needs to specify the number of thread blocks and the number of threads per block (maximum of 1024). Warps are always issued as groups of 32 threads and compute power will be wasted if the size of a block is not a multiple of 32. Every individual thread knows exactly where it fits in regards to the greater problem through the index variables `threadIdx` and `blockIdx`. Normally the first computation a thread will do is to calculate its position in the block and grid, and use this information to load and work with the appropriate piece of data. The block scheduling method that CUDA employs is called SPMD (single program, multiple data), and is designed to closely match the capabilities of NVIDIA GPU hardware.

The grid describes how the greater problem is divided up into blocks, and is the last level of abstraction that the programmer must consider. Memory and thread usage should be organised in such a way as to avoid poor memory coalescing. Poor grid planning can result in up to a five times drop in performance [15]. For example, if a program is to perform operations on an image made up of 2000×2000 pixels, because 2000 is not a multiple of 32 there will inevitably be a waste of compute power at some stage in the computation. This is because each grid is made up of blocks, which in turn are made up of some multiple of 32 threads, and there is no way to organise the grid to perfectly match the size of the image. Finding the right balance between all of these components to maximise performance is often a trial-and-error process.

The final important topic to cover is how memory is handled in CUDA. The way that memory is accessed and cached during kernel invocation is crucial to maximising GPU performance. There are several different types of memory, each with advantages and disadvantages, and are ordered from fastest to slowest. The fastest memory are registers with are inside the device and consist of only a few kilobytes per SM. Shared memory is the next fastest, and allows a programmer to selectively cache commonly accessed data onto 64 kilobytes per SM. The remaining types of memory are constant memory, texture memory, device (or global) memory, and finally host memory. Constant memory is cached and is a good option for variables that must persist between kernel invocations. Global memory is basically the amount of memory that the GPU is listed as having in its specifications, usually consisting of several gigabytes. Global memory is writable from both the GPU and the CPU, and is the mechanism by which data is transferred back and forth. Host memory refers to the memory available to the CPU. A GPU program will often be developed first using only the slow global memory. This allows kernel functionality to be designed correctly without having to worry about running out of memory. The next step in development is usually fine-tuning the software to employ faster memory types.

5.5 MEX & CUDA IMPLEMENTATIONS

The MATLAB version of VoxelMARSFun worked as required and this section describes how it was ported into a single threaded C++ function, which was subsequently adapted into a CUDA kernel for running on a GPU. Creating VoxelMARSFun for several architectures provides an opportunity to compare running times and decide which architecture is best suited to this algorithm.

One of the many powerful MATLAB features is called MEX (MATLAB Executable) which provides an external interface that allows custom C, C++ or Fortran subroutines to be compiled and linked into a MATLAB script (often referred to as an M-function). The custom function will appear to MATLAB as if it's no different to a built-in function. The main reason for using MEX is to substitute one or more bottleneck M-functions with custom MEX code that performs the same task but in less time.

MEX-functions are normally written in C, and the basic structure is listed below.

1. Check incoming data from M-function.
2. Allocate memory for outgoing data.

3. Perform C/C++/Fortran function.

4. Return outgoing data to M-function.

Before the M-function can make use of the custom C code, the MEX-function source file must be compiled along with any other required source files. This process requires a compatible C/C++ compiler to be installed on the computer. Assuming no errors occur during compilation, a new compiled file is created with a platform-specific file extension. Compiled MEX files are not cross-platform and must be created independently for every type of computer. An extra compilation step is required if the user wants to include CUDA C++ functions in their MATLAB code. CUDA source code must first be compiled into an object file by the `nvcc` compiler provided by NVIDIA. Once the object file exists, it is then compiled along with the MEX-function to create the platform-specific function file.

The C++ version of VoxelMARSFun (referred to as the MEX version in later sections) was a port of the MATLAB version. There are many useful commands in MATLAB that don't exist in C++, which required that the functionality be recreated from scratch. The function took the circular volume parameters from the user and returned a complete list of voxels and path lengths. The function computed one x-ray elevation at a time, and for each elevation it computed the data for one ring at a time using a standard `for`-loop. This implementation of the algorithm ran on a single CPU core and was integrated into MATLAB through the MEX interface.

The CUDA version of VoxelMARSFun (referred to as the MEX+CUDA version in later sections) is similar to the MEX version, except that the main function was written as a CUDA kernel instead of relying on a `for`-loop, where the data for each individual x-ray elevation was computed in its own thread. There were two main problems with this implementation: algorithm complexity and memory requirements. While the algorithm is not particularly complicated, it is made up of several branching statements which does not make for an efficient GPU program due to the way warps process threads. This algorithm can be thought of as task-based parallelism, which is not the type of parallelism that GPUs are designed for. The other problem related to how much memory each thread required. Each thread had to store the voxel information (including voxel indices, chord lengths, etc) for that entire x-ray elevation. For a volume of several hundred rings, the memory requirement could be tens of megabytes per thread. This meant that blocks containing fewer threads had to be used. The CUDA function frequently checked the amount of free memory available on the GPU and only scheduled the number of threads that would actually fit into memory. Because of the large amount of memory required, only the slow global memory was

suitable for this implementation. The only optimisation made was storing several small variables in constant memory. Overall, VoxelMARSFun was not well suited for implementation on the GPU. Performance tests that compare the three implementations are discussed in the next section.

5.6 PERFORMANCE TESTING

In order to accurately compare the MATLAB, MEX, and MEX+CUDA versions of the VoxelMARS-Fun, it was important to keep implementation aspects as consistent across as possible. By default, MATLAB represents numbers with the double data type, so doubles were used when developing the MEX versions of the algorithm. No portion of the algorithm required the high numerical precision provided by the double data type, and this decision will have handicapped the MEX+CUDA version because memory management was the primary limiting factor in its performance. The three implementations were tested by running them with identical parameters and the resulting look-up tables were checked for consistency. Each test was performed three times, with only the best time being recorded. All tests were run on system #2 (Table 1) located in the MARS laboratory, and the results are shown in Fig. 27.

The MEX implementation is clearly the winner in all tests, with roughly a 30$\times$ speed-up over the MATLAB implementation running over four threads, and a 4$\times$ speed-up over the MEX+CUDA implementation. The MATLAB implementation was included in Fig. 27 to demonstrate the potential speed-ups one can achieve by implementing algorithms as a MEX-function instead of a pure M-function.

The look-up tables generated by this algorithm are particularly large. Generating look-up tables for reconstruction tests presented in chapter 6 required a table for each of the 720 projection angles. The size of the look-up table for a volume containing 250 rings was 600 MB, and for a volume containing 500 rings was 2.6 GB.

5.7 DISCUSSION

The overall performance of the three implementations does not come as a surprise. Compared to the singled threaded MEX implementation, the poor performance of the MEX+CUDA implementation is due to the algorithm complexity and memory requirements described in section 5.5. The memory optimisation research presented by Zhao *et al.* is intriguing, and similar methods

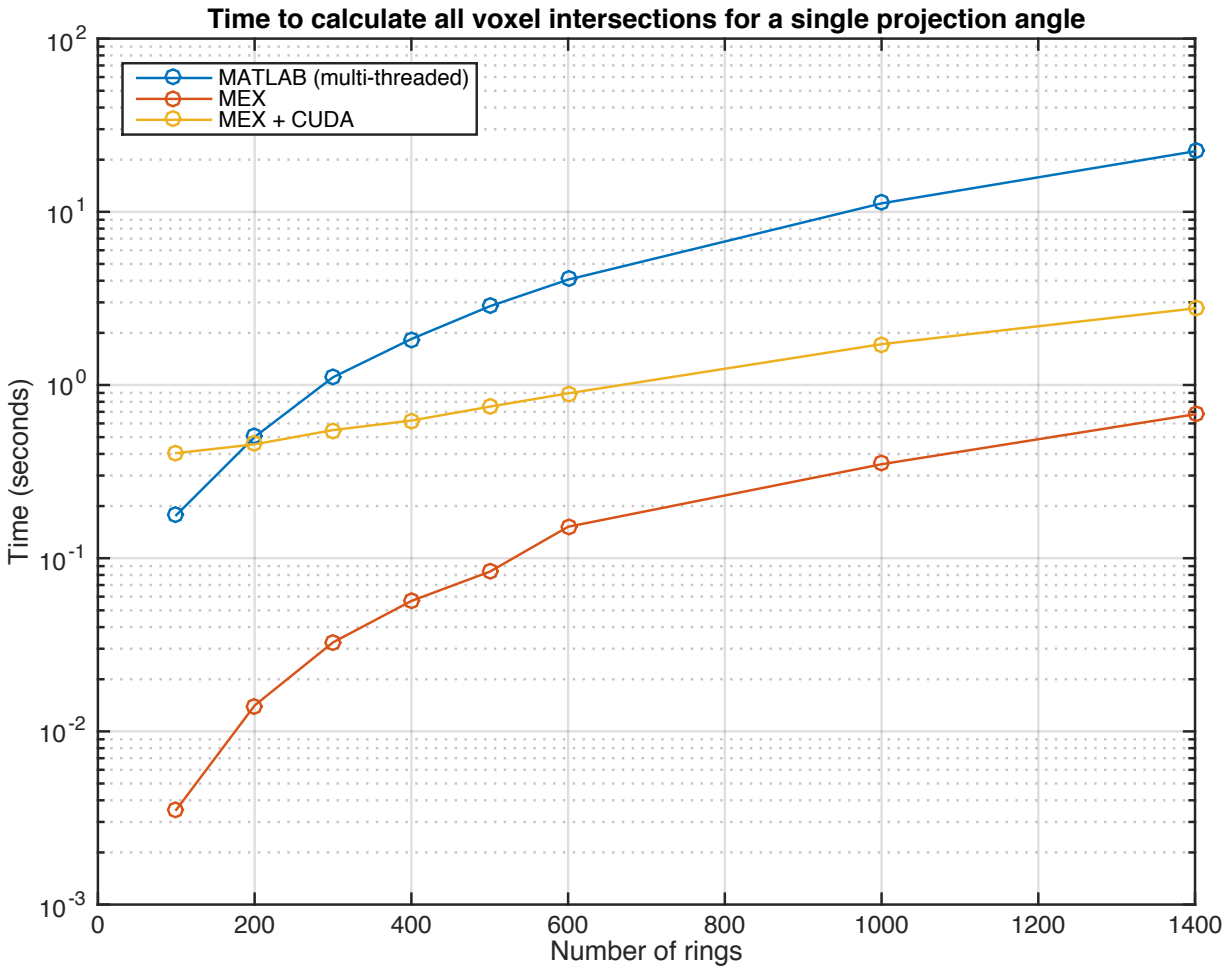


Figure 27: The MEX version of Alg. 5 is the most efficient implementation. The computational overhead associated with GPU computing can be seen by the poor performance of MEX+CUDA at very low numbers of rings.

may have to be implemented into the CUDA implementation of the MARS algorithm. There are still many aspects of the CUDA implementation that can be experimented with which may provide improved performance. The circular volume geometry was designed to treat every ring as a separate entity, where the path length is calculated on a ring-by-ring basis. The current CUDA implementation doesn't work that way, and it is instead parallelised in terms of x-ray elevation in the parallel-beam geometry (or x-ray angle in the fan-beam geometry).

A new algorithm will need to be devised which can instead parallelise the voxel intersection function in terms of the rings. Doing so will very likely reduce the complexity of the algorithm and eliminate many of the branching statements which don't work well on the GPU. There is obvious merit in making use of the many CUDA libraries that are available to the public. Similar to the way Flores *et al.* integrated the CUSPARSE library into their reconstruction software and

achieved a significant performance gain. The work done by Xin *et al.* is particularly encouraging, since they implemented SART as a GPU process, and SART is a similar reconstruction method to what is currently in development by the MARS group. Reducing the radiation dose is something that will be studied very closely as the human-sized MARS scanner begins development. The ideas presented by Jia *et al.* that show how to lower the dose by constructing a GPU-based reconstruction algorithm that can work with noisy and under-sampled data is something that may need to be integrated into the MARS image processing chain.

The amount of disk space needed to store the look-up tables was considerable. The large look-up tables indicate that moving to cone-beam geometries in the future will likely overload the memory available on a standard desktop computer, due to the inherent added complexity involved with adding a third dimension to the voxel intersection function. Luckily, the speed at which the MEX implementation computes the voxel intersections may invalidate the need to pre-compute look-up tables. Further refinement of the algorithm by including the extra formulation that eliminates the dependency on rotation angle, and integration into an ART-based reconstruction algorithm are the next steps in the implementation of the circular volume geometry.

The hardware and software development of the MARS scanner is an ongoing process and over the next few years developmental focus will shift towards scaling up the technology by a factor of 10^3 for the human-size scanner. The MARS scanner is also a commercial product that needs to be reliable and easy to use by customers. As of the end of this thesis, the current image reconstruction chain is made up of a group of distinct applications, which includes the image reconstruction software and MARS MD. Future development of the reconstruction chain will integrate all applications into one fluid software package that can be used by a client with minimal experience.

5.8 CONCLUSION

This chapter described how parts of the circular volume geometry model were taken into MATLAB, visualised, and implemented as a computer algorithm in several different languages. The Circular Volume Simulator tool allowed the user to directly compare two algorithms: one that was based entirely on MATLAB functions, and one that was based on the formulations introduced in chapter 4. The latter algorithm, VoxelMARSFun, was further developed as a single-threaded C++ function and as a CUDA kernel, both of which were run within MATLAB through the MEX

interface. The performance of the two MEX functions were compared in order to determine the appropriate computing architecture for future software development of the algorithm. The algorithm would be more appropriately run as a CPU function due to its high memory requirements and the fact that the GPU is not suited to the task-based parallelism model. The large look-up tables and the speed of the MEX implementation provides evidence to suggest that the calculation of voxel intersections should be performed on the fly during a reconstruction algorithm. Many researchers have had great success with implementing iterative algorithms on the GPU, so development of the CUDA implementation of the circular volume geometry will continue by exploring more advanced optimisation processes.

5.9 SUMMARY

- This chapter has presented work done during this thesis to develop efficient algorithms that calculate the voxel intersections of a given x-ray passing through a circular volume.
- The intersections and x-ray path lengths for every voxel are saved as a look-up table, which serves as the basis for the system matrix of an ART-based reconstruction algorithm.
- A visualisation tool called the Circular Volume Simulator was developed and considerably simplified the process of algorithm development by letting the user draw accurate volume representations to the screen.
- Implementations of the algorithm were written in MATLAB, C++, and CUDA. For consistent comparisons between implementations, all were executed through the MEX interface within MATLAB.
- Performance tests of all implementations were completed, and the single-threaded C++ version was the fastest. This was due to the high memory requirements and task-based parallelism that existed in the CUDA version.

6

VOLUME VISUALISATION

This chapter describes the motivation and methods behind mapping the circular geometry to square geometry, and the quality of the images produced after incorporating circular geometry into a well-known reconstruction method. Sections 6.1 and 6.2 discuss background to image processing and similar work conducted by independent researchers; section 6.3 explains the design choices and implementations of the chosen mapping method; section 6.4 outlines the methods and results of performance testing the different implementations; section 6.5 describes how the circular geometry performed compared to a traditional square voxel system; and the chapter concludes with a discussion in section 6.6, a conclusion in section 6.7, and a summary in section 6.8.

6.1 INTRODUCTION

The circular volume geometry and related algorithms allow voxels to be allocated with x-ray path length data very quickly. By matching the geometry of the reconstruction volume to the circular rotation of the MARS gantry, the number of dimensions of the system has been reduced by one. However, the circular geometry doesn't align with the square pixels of a computer screen, so when it comes to displaying a reconstructed image to the user, a transformation step must first be completed. There are many ways to complete this task, all with varying degrees of computational complexity. The algorithms designed in chapter 5 generated look-up tables that represented the system matrix of the reconstruction, and were first prototyped in MATLAB and later extended into C++ and CUDA. The same approach was followed in designing the algorithms in this chapter, with the goal of comparing the achievable degree of parallelism and ultimately choosing the appropriate hardware architecture for the problem.

This chapter describes two methods for drawing the circular volume to the screen, and both use a system whereby the entire circular volume is covered in a grid of evenly spaced Cartesian points. These points describe the transformation between the two geometries. The look-up table generated is a square array with the dimensions of the grid, and each point on the grid stores the linear index value for the voxel it represents. The first algorithm, called VoxelMapFun, works

on a voxel-by-voxel basis and decides which points lie within the voxel by creating a temporary grid of points over the voxel. This algorithm was extremely slow, which led to the development of the second algorithm called PointMapFun. This algorithm worked on a point-by-point basis, and employed simple trigonometry to calculate the voxel that each point corresponded to. Both algorithms generated identical look-up tables, and a simple function was written to load a look-up table and draw the volume to the screen. The entire process of creating a look-up table and drawing the volume to the screen was very fast, and a standard volume consisting of 500 rings only took about five seconds to complete.

Keeping with the theme of visualisation, the chapter concludes with a brief analysis of the circular volume geometry being used as part of a FBP reconstruction. The circular volume geometry will soon be integrated into the new iterative algorithm being developed by the MARS group, so the goal of this chapter was to attempt such an integration into a well-known reconstruction algorithm. For the purpose of testing the circular volume geometry it was decided that FBP be used as the reconstruction method. This made the process of integrating the circular volume geometry much simpler, and also meant that reconstructions were completed very quickly. The results indicated that a given reconstruction algorithm will need to be redesigned to some extent, in order to fully take advantage of the unique properties of the circular volume geometry.

The software development and analysis described in this chapter represents a substantial step forward in the integration of the circular volume geometry, and will significantly contribute to a faster and higher-quality image processing chain in future MARS scanners.

6.2 RELATED WORKS

Chapter 4 presented a review of several relevant circular volume geometry techniques reported in the literature. The authors briefly described the same transformation problem that exists for the circular volume geometry investigated in this thesis, namely having to draw the volume to the screen.

Rodríguez-Alvarez *et al.* introduced the polar voxel concepts URA-C and CR-P, and found that creating a system matrix with URA-C was efficient enough to eliminate the need for system matrix pre-calculation and storage. They noted that an image viewer must be devised to display the reconstructed images and minimise any deterioration that occured when transforming from polar coordinates to Cartesian coordinates [28]. They performed reconstructions using two sets of data:

real measurements obtained from a CT-simulator; and simulated data generated by the CTSim software package. For consistency, CTSim was configured to the same geometrical dimensions as the CT-simulator. All of their system matrix methods, including URA-C and CR-P, were tested with the FBP and maximum-likelihood expectation-maximisation (MLEM) reconstruction methods. The objects being reconstructed were the Shepp-Logan phantom and a phantom containing circles of different densities that they referred to as the "lesion phantom". The image viewer developed by Rodríguez-Alvarez *et al.* places each polar voxel onto a Cartesian-grid with high enough density of points to account for four-times the image resolution size. This transformation created images of acceptable quality. Another viewer had to be created for the URA-C method because the holes between voxels had to filled in, and was accomplished with a bi-linear interpolation method. The viewer took three seconds to convert a polar grid image into a 2048×2048 pixel image. Reconstruction quality of the images was compared using the root mean square error (RMSE), signal-to-noise ratio. and the contrast recovery coefficient (CRC). Differences between the reconstructions were barely perceptible and there were only small differences in RMSE. The RMSE for all images was on the order of 10^{-3}.

The PART algorithm described by Jian *et al.* was highly symmetrical, with each voxel corresponding to a fan area. The authors didn't explain the details of how these voxels were transformed from polar coordinates to Cartesian coordinates, but they do state that bi-linear interpolation was employed for this purpose [29]. The interpolation caused deterioration in the reconstructed images, and area-weighted methods were mentioned as being a potential better option for displayed images reconstructed with PART.

The solutions implemented by other research groups and similar to what is developed later in this chapter. While Rodríguez-Alvarez *et al.* didn't go into much detail in regards to their solution, it seems that they implemented a Cartesian grid system similar to what is developed in section 6.3. The solution presented by Jian *et al.* follows a different path and implements a bi-linear interpolation method, which is considered the next method to try if the grid overlay method isn't satisfactory. It was therefore interesting to note that their interpolation method contributed to poor image quality, and may not be as appropriate as previously thought.

Voxels in the circular geometry must be mapped onto a rectangular geometry before being displayed on a computer screen. This is a well understood problem and various methods exist that efficiently solve it. The only challenge was to find a solution that best fitted the circular geometry scenario, keeping in mind the characteristics of the geometry and the potential end-user requirements of the MARS scanner.

For this thesis a simple method of overlaying a grid of points onto the circular volume was implemented. The points acted as a mapping from the circular geometry to a traditional square geometry. The dimensions of individual voxels that lie on a Cartesian plane can be easily calculated because they only consist of two straight lines that pass through the origin and two circular arcs centred on the origin (Fig. 28).

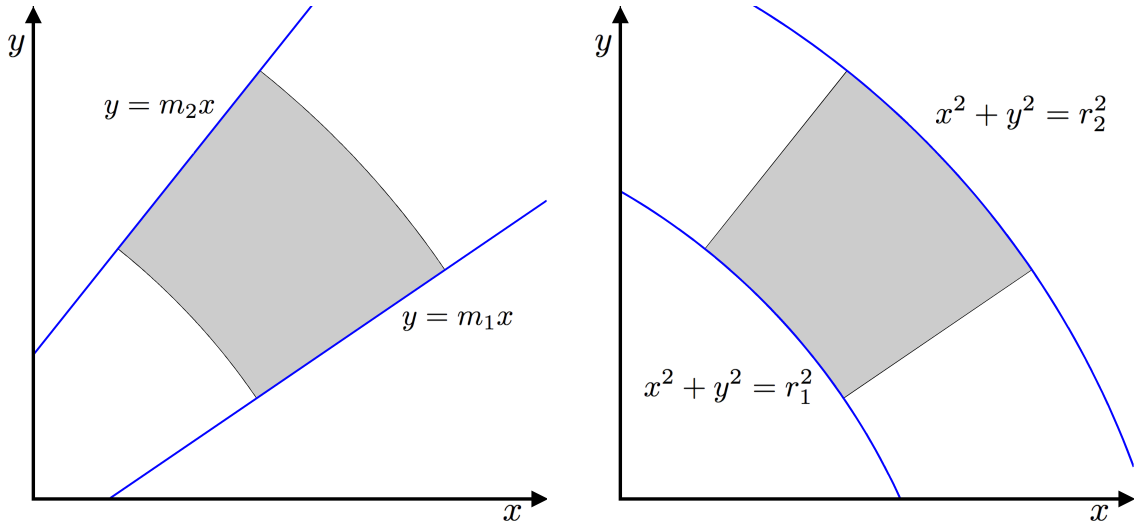
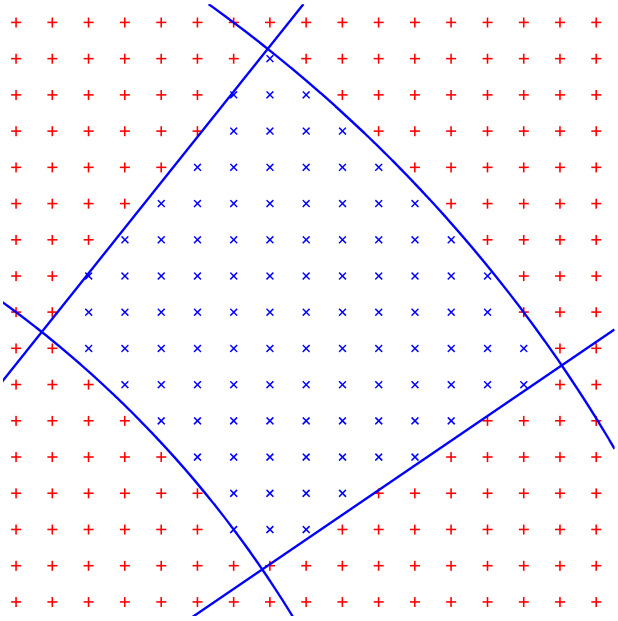


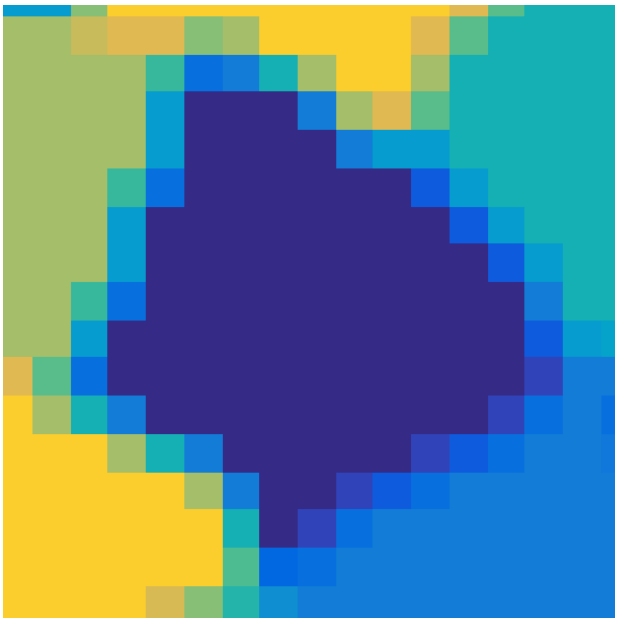
Figure 28: A 2D voxel is enclosed by two straight lines which intercept the origin and two circles which are centred on the origin. The area within the four edges can be classified with the two gradients m_1 and m_2 and the two radii r_1 and r_2.

The desired resolution of the image directly dictates the number of cartesian points in the mapping, and the algorithm directly assigns each point to either one voxel or to no voxel, if the point lies outside of the volume radius. The speed of the algorithm was considered only after it was shown to be working correctly. A simple mapping example is shown in Fig. 29a where the blue points are assigned to the voxel and the red points are not. Fig. 29b shows what a drawn voxel

might look like, where the individual squares can be thought of as pixels in this example. Each pixel is the mean of all voxel values that lie within the specified square grid of points.



(a) Cartesian points assigned to a voxel. A higher density of points will result in higher quality images, but will greatly increase the time required to generate the look-up table.



(b) Zoomed-in example of a rendered voxel with neighbouring voxels of different values. The jagged edges introduced by the mapping process are visible, and anti-aliasing effects are occurring at the edges of the voxel.

Figure 29: A simple point mapping procedure was implemented to transform the circular geometry into square pixels.

Once a look-up table exists for a particular volume, the next consideration is how the grid is drawn to the screen. If a grid consists of $10^6 \times 10^6$ points, for example, a method of downscaling is required before it can be displayed as an 800×800 pixel image. Distortion effects tend to occur whenever a high resolution image has to be displayed at a lower resolution. Various anti-aliasing techniques exist to correct this problem, and an example of this correction is visible in Fig. 29b where the pixel values located at the voxel edges appear as an intermediate colour due to the fact that a pixel is an average of all points inside of it. When the image is zoomed out far enough, these rough voxel edges will be less obvious to the viewer.

To draw a circular volume the following simple method was followed:

1. Calculate the number of cartesian points from the look-up table that will be included in each pixel.

2. Group all points into their corresponding pixel squares.

3. Calculate the mean voxel value for each square and assign the value to the appropriate pixel.

4. Draw the final image to the screen.

This method was preferred because images could be drawn at different pixel resolutions without having to re-create the look-up table. In order to reduce the likelihood of image artefacts caused by the conversion from circular to cartesian geometry, the level of detail used to perform the mapping was considered. The goal was to find the number of points per voxel that achieved a balance between good looking images (more points) and faster look-up table creation and image rendering (less points). In the end, using approximately 100 points per voxel achieved this goal. An example of an image created with this procedure can be seen in Fig. 29b.

6.3.1 *MATLAB Implementation: VoxelMapFun*

An algorithm called VoxelMapFun (Alg. 6) was written which individually analysed every voxel in a disk and determined which points on a Cartesian plane lay within it. The mapped points were saved to disk and were reloaded whenever an image was to be drawn to the screen. VoxelMapFun makes use of the trigonometric properties of each voxel to determine whether or not a specified point lies within the voxel. Both voxels and rings are independent of their neighbours, meaning that the parallelisation of this algorithm could have been designed in two ways. It made sense to parallelise this algorithm around the concept of the ring because the circular volume geometry is designed to calculate x-ray path length on a ring-by-ring basis. This choice also minimised the amount of transfer overhead to and from the workers in the MATLAB parallel pool, because there are significantly more voxels than rings in a volume.

VoxelMapFun operates on one voxel at a time, and in its parallelised form operates on four or more rings at once, with each thread operating on one voxel at a time. A key concept of writing parallelised code is to try and make sure all threads in a slice have a similar running time, because the worker must wait for all threads to complete before the next slice can begin. In VoxelMapFun this concept is mostly realised because neighbouring rings only differ by three voxels, meaning that there will only be a relatively short wait while the largest thread finishes.

VoxelMapFun essentially creates a bounding box around a voxel which is filled with candidate (x, y) points ready for assessment. Every point is checked by seeing if it lies between the two line gradients m_1 and m_2, and that it lies between the inner and outer ring radii, r_1 and r_2. If it

Algorithm 6: VoxelMapFun - Voxel indices look-up table.

Data: $disk, numPoints$
Result: $lookupTable$
foreach $ring \in disk$ **do**
  $numVoxelsInRing \leftarrow 6 \times ring + 3$
  $angleStep \leftarrow 2\pi / numVoxelsInRing$
  $angleVector \leftarrow 0 : angleStep : 2\pi$
  foreach $voxel \in ring$ **do**
    /* Calculate the gradients of enclosing lines (Fig. 28a) */
    $(\theta_1, \theta_2) \leftarrow (angles[voxel], angles[voxel + 1])$
    $(m_1, m_2) \leftarrow (tan(\theta_1), tan(\theta_2))$
    /* Calculate the radii of the enclosing circles (Fig. 28b) */
    $r_1 = (ring - 1) \times ringThickness$
    $r_2 = ring \times ringThickness$
    /* Use gradients and radii to find the points that lie outside the voxel */
    $(x_{min}, x_{max}, y_{min}, y_{max}) \leftarrow findExtremities(m_1, m_2, r_1, r_2)$
    /* Create bounding box of points over the voxel */
    $boundingBox \leftarrow$ meshgrid$(x_{min} : numPoints : x_{max}; y_{min} : numPoints : y_{max})$
    foreach $x, y \in boundingBox$ **do**
      /* Check the radius and gradient of (x, y) */
      $r \leftarrow \sqrt{x^2 + y^2}$
      $m \leftarrow y / x$
      if $(m_1 \leq m < m_2)$ & $(r_1 \leq r < r_2)$ **then**
        $includedPoints \leftarrow (x, y)$
      end
    end
    /* Index conversion by Alg. 2 */
    $index \leftarrow$ ConvertToLinear$(ring, voxel)$
    /* Store voxel index in global array at every included point */
    $lookupTable[includedPoints] \leftarrow index$
  end
end

meets that criteria, that point in the look-up table is labelled with the linear index of the voxel in question. If not, it is simply ignored. During the initial development of the circular geometry, operating on one voxel at a time allowed for easy visualisation of small sections of the volume. However, it wasn't designed with speed in mind and assessed grid points with low efficiency. The large number of points that lay outside of the voxel were simply rejected, completely wasting a significant proportion of the computation time because that same point will still have to be investigated by at least one other thread. Due to the highly inefficient nature of this algorithm, it wasn't deemed worthy of being rewritten as a MEX-function, but was still extremely useful as a diagnostic tool when designing the improved algorithm called PointMapFun. Creating look-up

tables using VoxelMapFun for 200 rings took about 15 minutes, and for 500 rings took well over an hour. A delay of this magnitude is unacceptable in a commercial product, so the development of the improved PointMapFun was essential.

6.3.2 *MATLAB Implementation: PointMapFun*

VoxelMapFun worked correctly, but for diagnostic purposes operated on a voxel-by-voxel basis. In an attempt to improve on that design, a new point-by-point algorithm called PointMapFun (Alg. 7) was developed that operated significantly faster. This method is also far more parallelisable that the previous one, and indeed the MEX and CUDA implementations performed very well.

At first glance it seems that PointMapFun might actually perform more calculations than VoxelMapFun and will therefore be a lot slower. For example, in a disk containing 500 rings and using approximately 100 points per voxel, the variables `numVoxelsInRing` and `angleStep` are calculated only 500 times each in VoxelMapFun, and a massive 75×10^6 times each in PointMapFun. This increase in computation is more than offset by the reduction in other types of computation, for example: fewer angles and distances to calculate; no need to find the minimum and maximum edges of a voxel; and no need to create a bounding box filled with points to be tested. The fact that there is no longer a need to waste time rejecting points outside of the bounding box is another time saving feature. PointMapFun ran an order of magnitude faster than VoxelMapFun.

A factor that needs to be discussed is the `meshGrid` input variable, which is actually two square arrays, one for every x coordinate and one for every y coordinate to be checked. There is a great deal of redundancy in these arrays, because for an $n \times n$ array the x array is made up of the same row of x values duplicated n times, and the y array is made up of the same column of y values duplicated n times. In MATLAB the creation of these mesh-grids is incredibly efficient, and allows for simpler code because the exact same array index used to determine x and y is also used to fill in the look-up table array. This type of simplicity makes the slicing of variables in a `parfor`-loop a great deal easier, however it also burdens the system with a significant waste of memory. When the program runs entirely on the CPU, memory management isn't so much of an issue because the operating system can efficiently deal with it. On a GPU, however, memory management must be handled by the programmer. This problem is addressed further in section 6.3.3.

Algorithm 7: PointMapFun - Voxel indices look-up table.

Data: $meshGrid, numRings$
Result: $lookupTable$
foreach $(x, y) \in meshGrid$ **do**
    /* Calculate distance from the origin to (x, y) and hence the ring number (assuming ring thickness of 1) */
    $ringIndex \leftarrow \left\lfloor \sqrt{x^2 + y^2} \right\rfloor$
    if $ringIndex \geq numRings$ **then**
        /* Point is located outside of the volume */
        $lookupTable[x, y] \leftarrow 0$
    else
        $numVoxelsInRing \leftarrow 6 \times ringIndex + 3$
        $angleStep \leftarrow 2\pi / numVoxelsInRing$
        /* Calculate the angle and hence the voxel number */
        $\theta \leftarrow \arctan(y/x)$
        $voxelIndex \leftarrow \lfloor \theta / angleStep \rfloor$
        /* Index conversion by Alg. 2 */
        $index \leftarrow \text{ConvertToLinear}(ringIndex, voxelIndex)$
        /* Store one-based voxel index in global array at (x, y) */
        $lookupTable[x, y] \leftarrow index$
    end
end

6.3.3 *MEX & CUDA Implementation*

This section describes how the superior VoxelMapFun was ported to C++ and CUDA. The relative simplicity of PointMapFun (compared to VoxelMapFun) made this a straightforward process. The function only consisted of 30 lines of code and there were no MATLAB functions that had to be rewritten. Similar to the design choices made in section 5.5, the overall design of the C++ algorithm were kept as close as possible to the MATLAB algorithm and the C++ and CUDA implementations were run in MATLAB through the MEX interface.

The MEX-function performs basic and preliminary checks on the data sent from MATLAB, and passes everything along to the linked C++ subroutine to perform the actual calculation. One of the most important functions that the MEX-function performs is the memory allocation of the output variable (the look-up table) that will eventually appear in the MATLAB workspace. The MEX-function also passes the associated memory pointers of the look-up table array and the two (x, y) mesh-grid arrays through to the linked C++ subroutine. The arrays themselves don't need to be copied because only the memory pointer gets passed between the MEX-function and the

C++-function. As a result, the entire algorithm can be performed in a single iteration made up of a nested `for`-loop. Performance results are displayed in Fig. 31.

The conversion from standard C++ into CUDA was also relatively straightforward. The MEX-function remained largely unchanged, but as previously discussed, the compilation process now required two steps. A small enhancement made in the CUDA implementation involved storing variables that were common to all threads, such as `ringThickness` and `totalNumberOfVoxels`, as a *constant* variables for fast access. It should be noted that the same mesh-grid functionality was copied through into the C++ and CUDA implementations. This design choice caused a significant memory problem, and led to having to take an active role in memory management. As a result, there are two CUDA versions of the algorithm to consider. The first version, referred to as MEX+CUDA, performs no specific memory management techniques, and as such operates exactly the same way as the MATLAB and MEX versions of the algorithm, meaning that the two entire mesh-grids were created in MATLAB and used to produce the look-up table. For small numbers of rings (less than 500) this procedure worked with no problems. As the amount of data increased, however, the GPU could no longer store the entire arrays in memory. This limitation required the MEX-function to take a more active role in dealing with how much data to send to the GPU in each iteration, with the computation having to be broken up into sections. The MEX-function would perform a task similar to that of slicing up an array in a MATLAB `parfor`-loop, where it would ask the GPU how much memory was available at the start of each iteration (using the `cuMemGetInfo` function), and then select the p array columns that would fit into the available memory. It is never safe to assume that the same amount of memory will be free at the start of each iteration, particularly if the GPU is also powering a computer screen, because the operating system can make background requests of the GPU at any time. This issue occurred frequently during development of the algorithms (using system #1). A printout of free memory at the start of each iteration often showed wild fluctuations.

With three identically sized $n \times n$ arrays needing to fit into GPU memory, the value of p was determined at each iteration by:

1. Determine GPU free memory $\rightarrow M$
2. Determine the amount of memory required for a single column of data ($\times 3$ for the three arrays) $\rightarrow N = 3 \times n \times$ `sizeof(double)`
3. Calculate $\rightarrow p = M/N$

The advantage of this method is that for every thread performing the calculation, the relevant array element in each of the three arrays has the same `threadIdx`, because the arrays have identical dimensions at every iteration (Fig. 30a). The disadvantage of this method is that there is still a huge amount of redundant information being copied to the GPU. This was a convenient design for the early stages of development, but eventually had a noticeable negative impact on program performance. In section 6.4 it can be seen that the MEX+CUDA function runs at an almost identical speed to the single-threaded MEX function. This outcome led to the development of the next CUDA version, referred to as MEX+CUDA (enhanced).

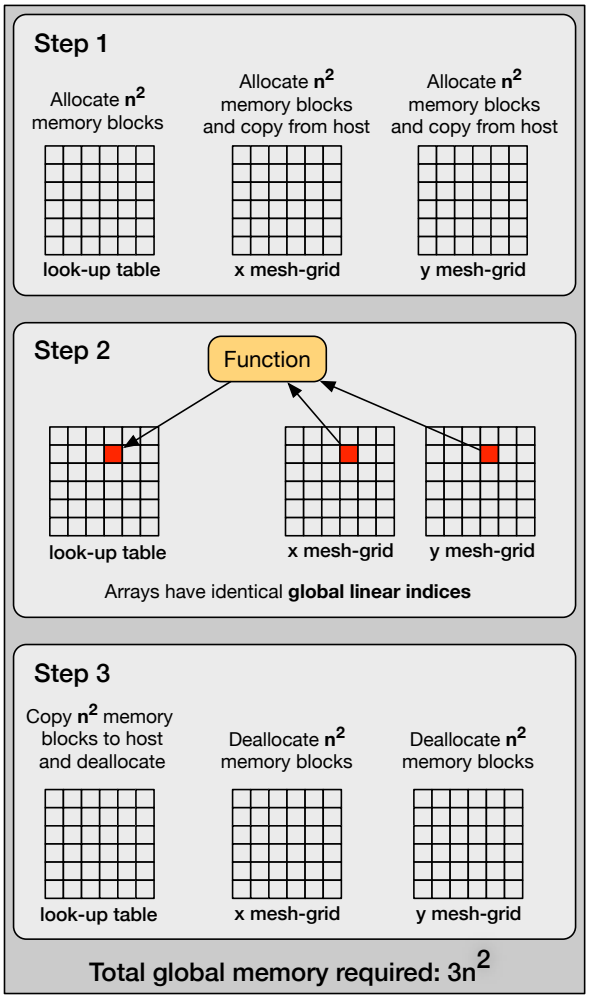
The only change made in the enhanced MEX+CUDA function deals with the redundancy introduced by the two large mesh-grid arrays. Instead of copying the mesh-grids into GPU memory, it made more sense to simply copy the n-length vectors that the two mesh-grids are based upon. They take up far less memory and only need to be copied once and can be re-used as many times as needed. The CUDA subroutine had to be altered to index the x- and y-vectors instead of the full mesh-grid arrays. Because the algorithm was split up into sets of columns, the `threadIdx.y` variable was still usable as the y-vector index due to the fact that the $n \times p$ look-up table array shares the same dimensionality in the y direction as the y vector (with n elements). The only problem encountered was with the `threadIdx.x` variable along the x dimension. Even though the entire n-element x-vector is stored in GPU memory, only a portion of it is needed at each iteration. As such, it was necessary to maintain a variable to keep track of how many columns have been processed up to this point, and then an offset is added to `threadIdx.x` in order to find the correct x-vector index.

Fig. 30 shows the differences between the two approaches. The huge reduction of required GPU memory in Fig. 30b is one of the key reasons for why this algorithm works so much more efficiently.

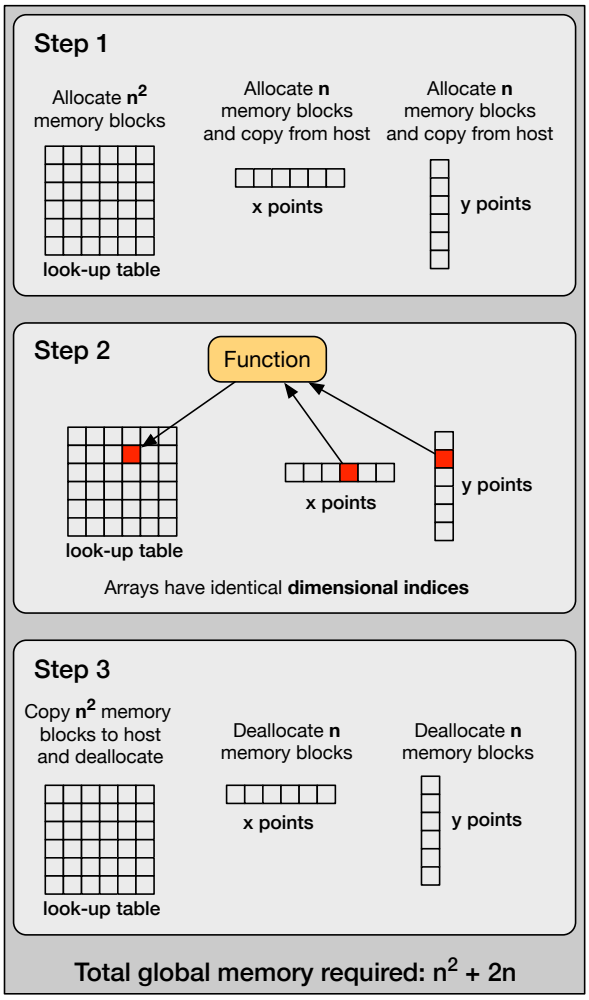
The small image processing algorithm that loads a given look-up table and draws the circular volume to the screen was also re-written as a standard MEX-function. The old MATLAB version took over a minute to draw the volume, while the MEX version took less than five seconds.

6.4 PERFORMANCE TESTING

The four implementations of the algorithm were tested by running them with identical parameters and the resulting look-up tables were checked for consistency. Each test was performed three



(a) MEX+CUDA: Allocating two entire mesh-grid arrays on the GPU for every iteration of the loop resulted in performance that was comparable to the single threaded CPU implementation.



(b) MEX+CUDA (enhanced): Making smart memory management decisions like allocating the x and y points as single 1D arrays led to a significant performance improvement over the single threaded CPU implementation.

Figure 30: The GPU has a limited amount of global memory that must be carefully managed by the developer. The two methods of memory management attempted in the CUDA implementation of PointMapFun are outlined.

times, with only the best time being recorded. All tests were run on system #2 (Table 1) located in the MARS laboratory, and the results are shown in Fig. 31.

While VoxelMapFun and PointMapFun generated the same look-up tables, the time taken to achieve the result was drastically different. PointMapFun is an example of a function that can be easily run on a GPU, because there is a direct one-to-one correlation between input and output arrays, every array element is independent of its neighbours, and the algorithm itself is an example of data-based parallelism. VoxelMapFun was not part of the performance testing suite. The

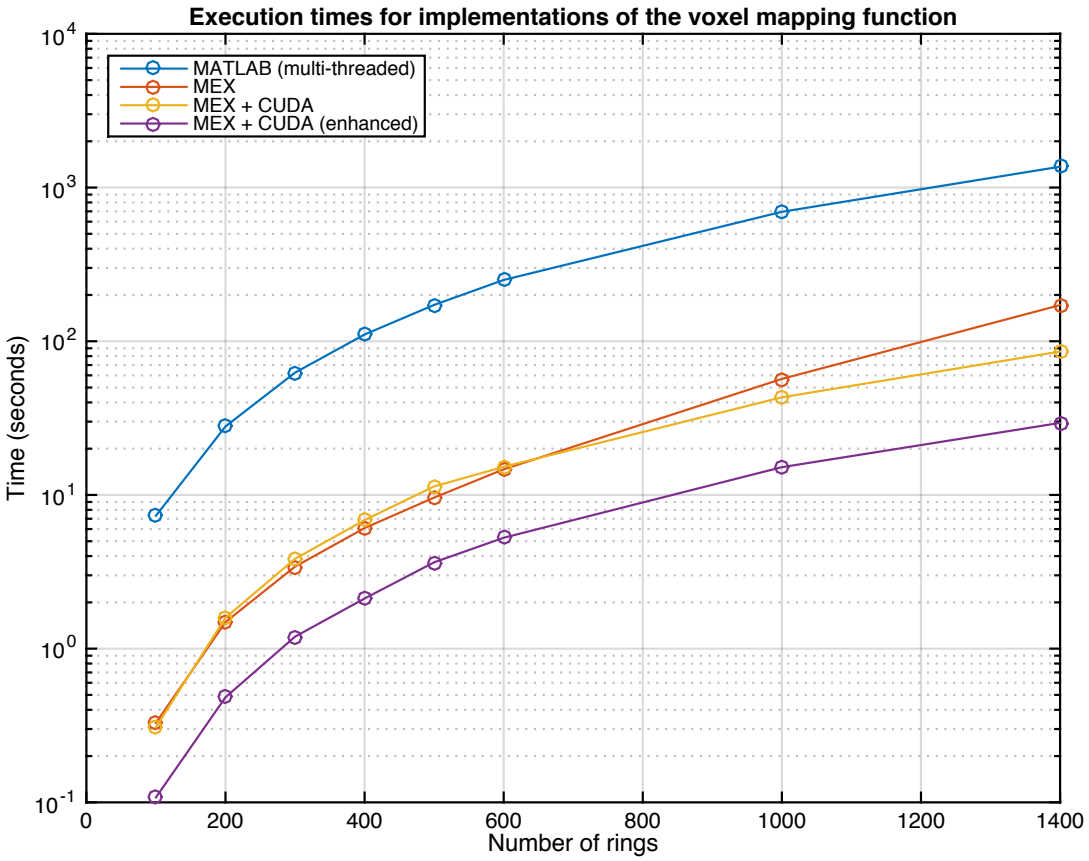


Figure 31: The highly parallel nature of PointMapFun results in an impressive speedup on the GPU when compared to the same algorithm running in MATLAB code and standard C/C++ code through the MEX interface.

MATLAB implementation was included in Fig. 31 to demonstrate the potential speed-ups one can achieve by implementing algorithms as a MEX-function instead of a pure M-function.

There are several points of interest that can be seen in Fig. 31. The first is that the MEX+CUDA (enhanced) function runs between 2-6$\times$ faster than the equivalent MEX+CUDA function, with the only change made being to the way the x and y points were supplied to the subroutine. The number of points to be copied was reduced from $2n^2$ to $2n$, and the data only had to be copied once. This outcome illustrates the importance of minimising the reliance on CPU-GPU data transfers. Also, because the memory requirement per iteration was essentially divided by three, this allowed more array columns (and hence more threads) to be computed per iteration. The MEX+CUDA (enhanced) implementation ran between 40-60$\times$ faster than the multi-threaded MATLAB implementation.

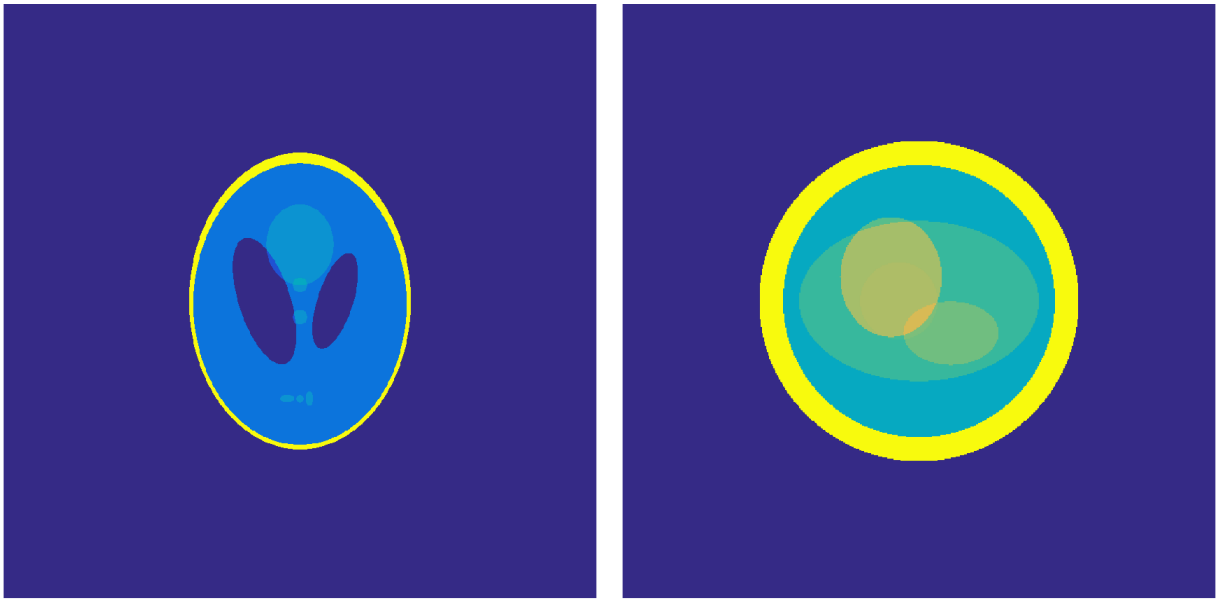
One of the primary goals of developing the cylindrical volume geometry is to assess its suitability for inclusion into a real-world image reconstruction processing chain, such as that used on a MARS scanner. The current reconstruction algorithm employs cubic voxels and requires a large portion of time dedicated to path length calculations. This raises an important technical question: is a given reconstruction algorithm reliant on the geometries of its voxels, or can different geometric models be easily inserted into the algorithm with no change to the overall functioning of the reconstruction? The aim of this section is to attempt to answer this question. The idea was to take a well-known reconstruction algorithm and alter the section of code that deals with voxel selection to instead process a circular volume instead of a square volume. The remaining code was not changed. Once the algorithm was able to complete a reconstruction with the circular geometry, direct visual and statistical comparisons were made between the images produced by the traditional square geometry algorithm and the circular geometry algorithm. The algorithm chosen for this task was filtered back-projection (FBP) and the template MATLAB scripts used were written by Mark Bangert [35]. These scripts were an excellent learning resource because they could be used to compare the images produced from several different techniques: simple back-projection; FBP in the spatial domain; FBP using 2D Fourier transformations; and FBP using 1D Fourier transformations and the central slice theorem. As expected, all of the techniques gave very similar results, except for the simple back-projection. The FBP spatial domain algorithm was ultimately chosen for the full range of tests because it performed marginally better then the others.

6.5.1 *Phantoms & Testing Conditions*

The FBP reconstruction tests were undertaken using two different phantoms: (1) the Shepp-Logan phantom (Fig. 32a); and (2) a phantom employed by Feldkamp *et al.* [36] (Fig. 32b), which will be referred to as the Feldkamp phantom for the remainder of this chapter.

The Shepp-Logan phantom is the most recognisable and commonly used phantom in the field of Medical Imaging. It was developed by Larry Shepp and Benjamin F. Logan in 1974 as a model of the human head. The Feldkamp phantom is a three dimensional phantom constructed from a set of superimposed ellipsoids which are placed asymmetrically throughout the phantom. The central

slice was chosen for all testing scenarios because it contained a variation of shapes and densities. The outer-most rings of both phantoms are representative of the high density of bone, which is a good test of the accuracy of a reconstruction algorithm, because high density material can often dominate during reconstruction and prevent less dense materials from displaying correctly. The high-density outer ring of the Feldkamp phantom is substantially thicker than that of the Shepp-Logan phantom, and should provide a good way of characterising how the reconstruction algorithm handles high-density material.



(a) Shepp-Logan phantom (b) Centre slice of the Feldkamp phantom

Figure 32: Phantoms used for FBP reconstruction testing.

The Shepp-Logan and Feldkamp phantoms were created at resolutions of 512×512 pixels and 256×256 pixels, respectively. The chosen resolution difference is due to the fact that the Feldkamp phantom took a long time to be constructed at increasing resolutions, but it was also worthwhile testing the algorithm and volume geometries at different resolutions. The reconstruction algorithm takes the pre-defined phantom in a square array and forms a much larger square around it by padding the edges with zeros. This step is necessary because at every projection angle the phantom is rotated about the central axis, but must still be entirely enclosed within the same sized square array otherwise data will be lost. Thus, the 512×512 pixel image became 942×942 pixels, and the 256×256 pixel image became 474×474 pixels. The sinograms were generated using the MATLAB function `imrotate` which rotates a square array by using linear interpolation techniques to calculate the new positions of every element in the array.

Reconstruction occurred over 720 projection angles (one full rotation with 0.5° increments). The sinogram image is made up of projections through every row in the phantom image, taken at every projection angle. In the case of the Shepp-Logan phantom, the sinogram was a 942×720 pixel image and for every projection angle 942 back-projections were performed. The square geometry reconstruction is straightforward because every back-projection corresponds to a set of equally spaced and equally sized square voxels, although the number of voxels will differ depending on the projection angle. In order to keep the square and circular reconstructions as similar as possible, it was important to try and keep the size of the voxels as similar as possible. This meant choosing a ring thickness that matched the height and width of the square voxels, and therefore the number of rings had to equal half of the number of rows in the sinogram, since each ring is represented both above and below the centre line. The number of rings was therefore 471 and 237 for the Shepp-Logan and Feldkamp phantoms, respectively.

A common method used to compare two equally sized images is the root mean square error (RMSE) (Eq. 6.5.1), and was employed by Rodriguez *et al.* (2011) during their reconstruction testing [28]. The same method is applied to the reconstructions conducted in the remainder of this section. The RMSE is given by

$$\text{RMSE} = \frac{1}{N} \sum_{i=1}^{N} ||x_i^r - x_i^0|| \tag{6.5.1}$$

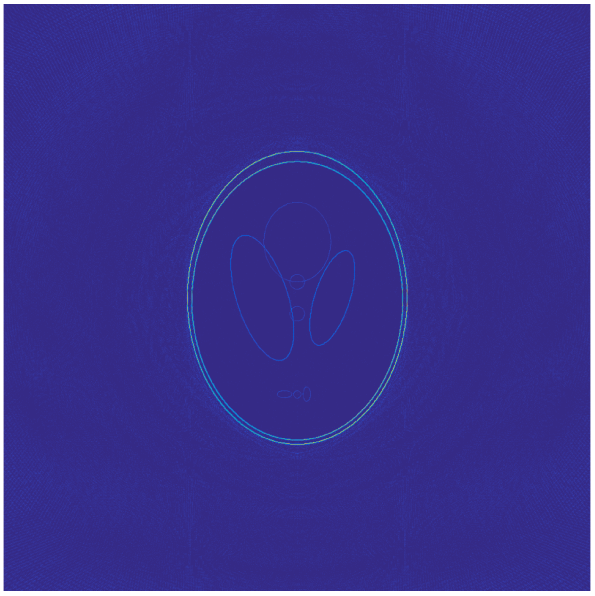
where N is the number of pixels in the image, x^r is the reconstructed image, and x^0 is the phantom image.

6.5.2 *Square Geometry*

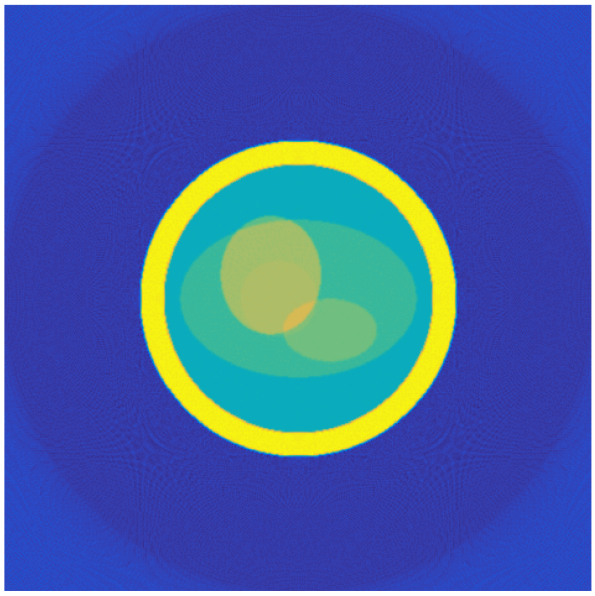
The square geometry used by default in the FBP algorithm written by Mark Bangert [35] and uses a mesh-grid method to calculate the intersected voxels at each projection angle. Two mesh-grids, one for the x dimension and one for the y dimension, were made up of integer coordinates from the centre of the reconstruction volume. The algorithm decides which voxels relate to each ray by adding the two mesh-grids after multiplying the x-grid by the sine of the rotation angle and the y-grid by the cosine of the rotation angle. This calculation is only approximate because the voxel distances always have to get rounded to the nearest integer. The rounding error doesn't have a noticeable negative effect on the reconstruction process for two reasons: (1) the FBP algorithm works well because the sinogram information can be evenly spread across the evenly

spaced and sized voxels in the square geometry; and (2) due to the approximate nature of the FBP algorithm, it provides a natural blurring effect which helps to hide any errors introduced by voxel approximations.

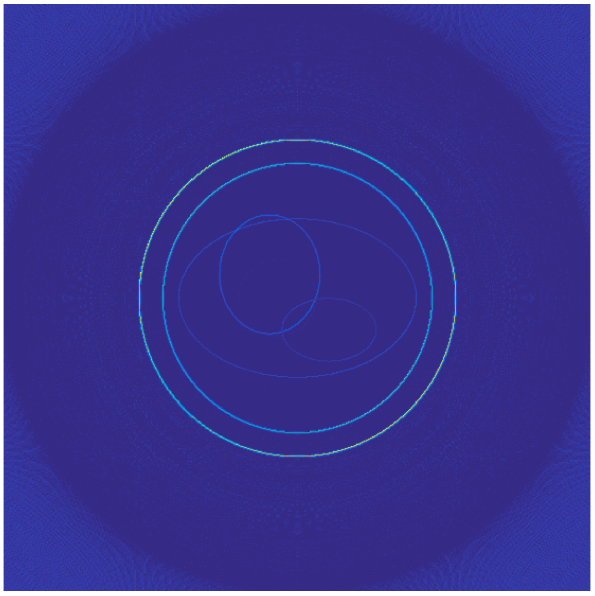
(a) Square geometry reconstruction image with a normalised RMSE of 0.0746.



(b) Absolute difference between phantom and square geometry reconstruction.



(c) Square geometry reconstruction image with a normalised RMSE of 0.0654.



(d) Absolute difference between original phantom and square geometry reconstruction.

Figure 33: Performing filtered back-projection image reconstruction of both phantoms with 720 projection angles using the square geometry. The only noticeable image artefacts are the predictable blurred edges that occur at object boundaries.

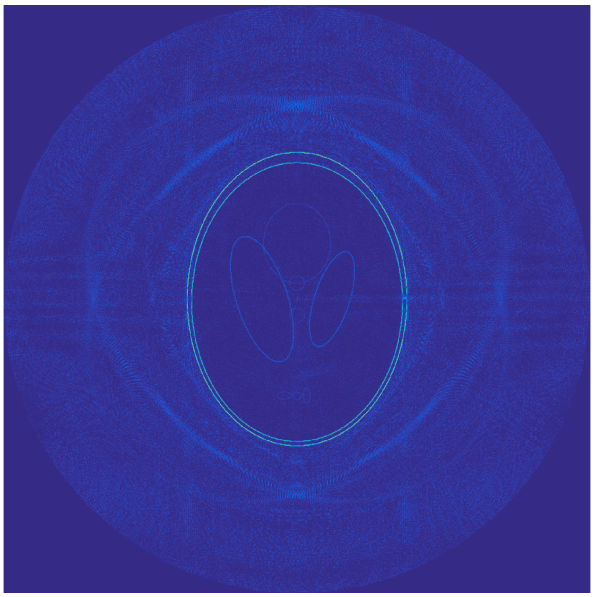
Figs. 33a and 33c are examples of what to expect from a good back-projection algorithm. Taking the difference of the normalised phantom and reconstruction images (Figs. 33b and 33d) helps to highlight the inaccuracies in the final images. In this case, there is a noticeable blurring effect at the border of distinct objects within the phantoms caused by the sudden change in material density. Blurring effects like this are common in FBP reconstructions.

6.5.3 *Circular Geometry*

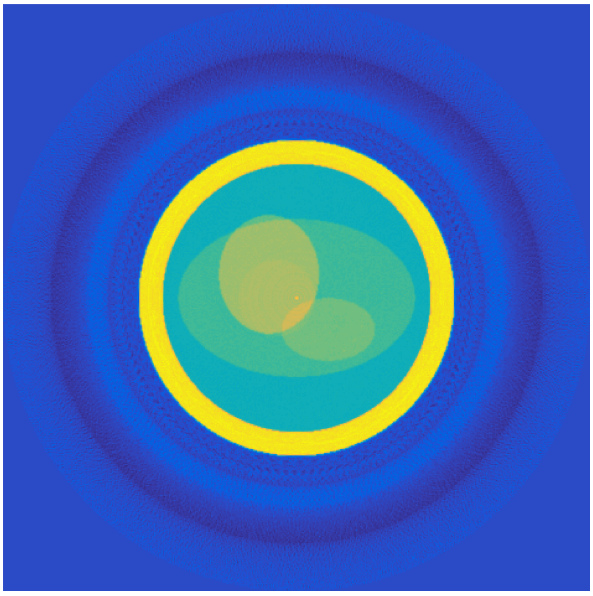
The second set of reconstructions were performed using the system matrix data created for the circular volume geometry, as described in chapter 5. Voxel and path length look-up tables for volumes containing 471 and 237 rings were created for the reconstructions, which took up 2.3 GB and 562 MB of disk space, respectively. The only change made to the FBP algorithm was in the way voxel indices were determined for each projection angle and detector elevation. Instead of performing an approximate rotation of the phantom image in order to determine which voxels lay in the correct path, it was simply a matter of loading the look-up table file for the current rotation angle and selecting the pre-calculated voxel indices that lay along the photon path. The results are shown in Figs. 34a and 34c.

Figs. 34a and 34b show strange spiral artefacts in the Shepp-Logan phantom, and Figs. 34c and 34d have a similar artefact but in a circular pattern in the Feldkamp phantom. On closer inspection it seems that the size and shape of these artefacts is related to the size and shape of the high-density outer section of each phantom. The oval shape of the Shepp-Logan phantom has been translated into the artefacts at 45° to the left and the right of the centre line. A similar effect happens with the Feldkamp phantom, but since the high-density section is a perfect circle, the artefacts likely overlay each other. RMSE for the Shepp-Logan and Feldkamp phantoms are 0.1146 and 0.0852, respectively, much higher than the 0.0746 and 0.0654 values reported for the square geometry reconstructions. The phantoms themselves have been accurately reconstructed, with the same blurring effects at the edges that were seen in Fig. 33. It is the outer region of the images that appears to be extremely turbulent. One may argue that this region of the image represents air or empty space and so can be discounted as being not important, but the turbulence is likely also happening inside the phantom and must still be carefully considered. Another artefact can be seen in the horizontal streaks that run across the centre of both images.

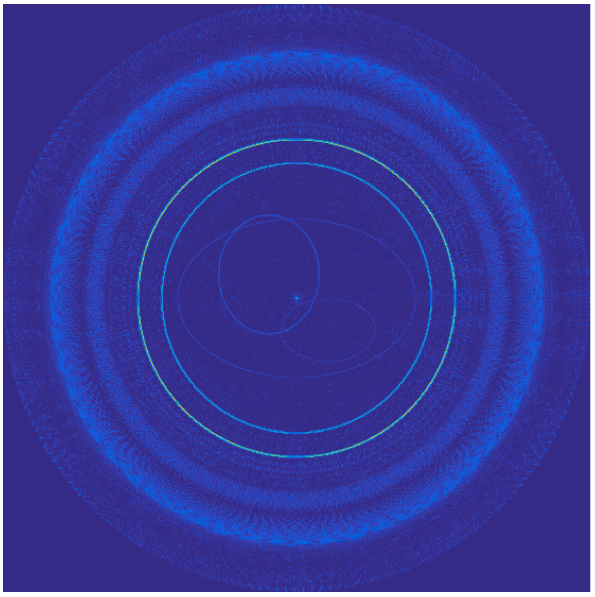
(a) Circular geometry reconstruction image with a normalised RMSE of 0.1146.



(b) Absolute difference between phantom and circular geometry reconstruction.



(c) Circular geometry reconstruction image with a normalised RMSE of 0.0852.



(d) Absolute difference between original phantom and circular geometry reconstruction.

Figure 34: Filtered back-projection images generated with the circular geometry. There are noticeable image artefacts present in both sets of images, likely caused by assumptions made about voxel shapes within the algorithm.

6.5.4 *Circular Geometry with Path Length Correction*

The problem with the reconstruction from the previous section is that the algorithm still assumes each voxel has the same dimensions and are placed next to each other with no gaps in between.

It has been shown that this scenario does not exist in the circular geometry. While there is no simple way to correct for the gaps in the image, there is a simple method to partially correct for the voxel length assumption which involves making use of the path lengths calculated and stored in the look-up table. At each projection angle, before the back-projection data is copied into the assigned voxels, the individual data for each voxel is multiplied by a correction factor that corresponds to the overall contribution that the voxel makes to the path length through the volume. For example, if the average path length through a voxel is 1 unit, but a chosen voxel instead has a path length of 0.9, then the back-projection data for that voxel is multiplied by 90%. This method is extremely approximate and not intended to be a full solution to the image artefacts seen in the previous section, but by finally involving the path length values into the back-projection formula, there should be some improvement to the final images. Figs. 35a and 35c show the new set of images generated with this correction factor.

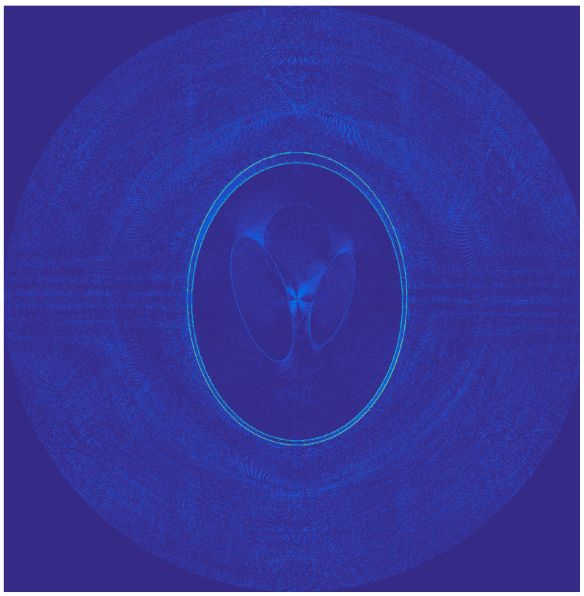
At first glance, the images appear to be much improved since the noticeable spiral or circular artefacts have mostly disappeared. However, Figs. 35b and 35d show in more detail that there are artefacts in the centre of the images. RMSE for the Shepp-Logan and Feldkamp phantoms have also increased to 0.1451 and 0.1781, respectively. Not only are the horizontal streaks across the images still present, but there appears to be new streaking artefacts that emerge from the very centre of each phantom. This effect may be caused in some way by the highly non-square voxels that live in the centre of the volume, although this is purely speculative.

6.6 DISCUSSION

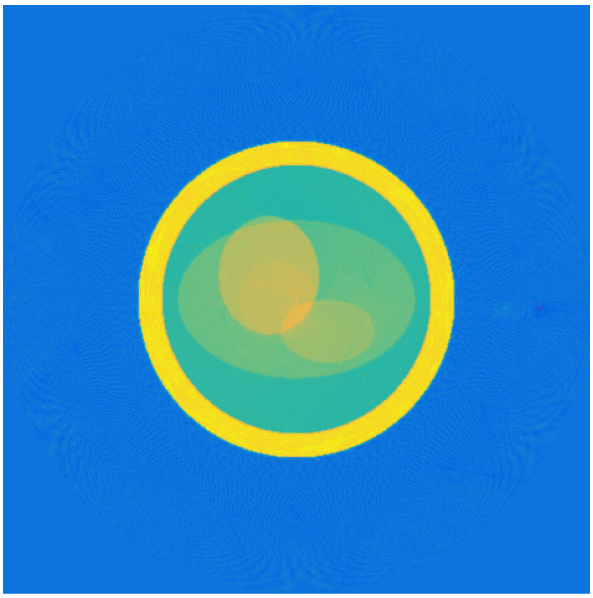
The development of the grid overlay mapping algorithm was influenced by several design goals, all of which were successfully achieved. The design goals were:

- The algorithm should be able to accommodate any number of rings, disks, and volume dimensions.
- Once a complete mapping has been generated, it can be saved to disk and used as a look-up table.
- Implement the algorithm in both C++ and CUDA and compare the execution speeds to determine any benefit/detriment of parallelisation.
- Make the algorithm fast enough to eliminate the need for a look-up table.

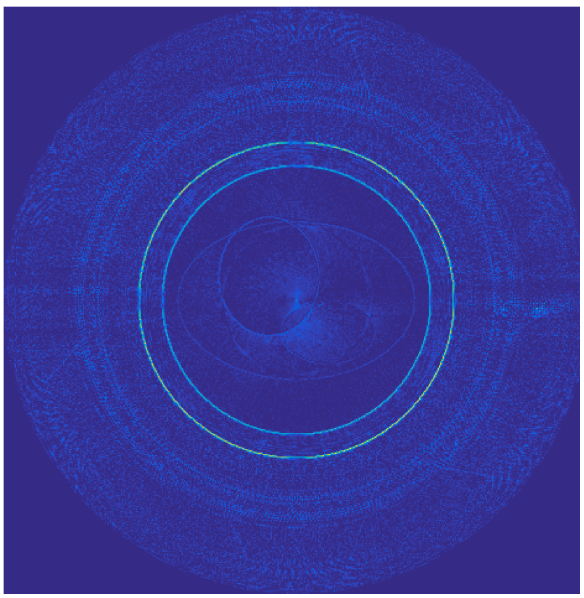
(a) Circular geometry reconstruction image with a normalised root-mean-square error of 0.1451.



(b) Absolute difference between phantom and circular geometry reconstruction.



(c) Circular geometry reconstruction image with a normalised root-mean-square error of 0.1781.



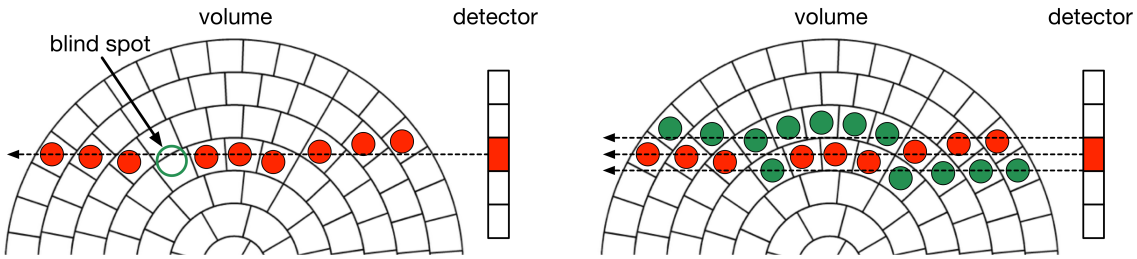
(d) Absolute difference between original phantom and circular geometry reconstruction.

Figure 35: Filtered back-projection images generated with the circular geometry and a path length correction factor applied. The spiral and circular artefacts have mostly disappeared, but have been replaced by artefacts in the centres of the images that are reminiscent of beam hardening artefacts.

In relation to the performance testing results, it is certainly possible that a multi-core implementation of the MEX function will provide even faster performance than the CUDA function and is something that may be investigated in future research. However, the preliminary results

shown in this chapter point to the obvious interpretation that GPUs are more efficient at graphics processing than CPUs. There are no doubt many CUDA optimisations that can still be made that will provide an even greater performance boost.

One of the aims of this chapter was to attempt to plug in the circular geometry into an existing reconstruction algorithm that is designed for square geometries, and to see if the algorithm still works as expected. After viewing the artefacts in Figs. 34 and 35 it seems clear that a given reconstruction algorithm does depend to a certain degree on the shape and distribution of its voxels. FBP is designed around the concept that the sinogram data will be spread across a string of equally spaced square voxels, a situation which is impossible with the circular volume geometry.



(a) A typical problem associated with the voxel intersection method is manifested as a "blind spot" that can occur with unfortunate combinations of ring number and photon elevation.

(b) One potential solution to this problem involves combining the intersected voxels obtained from ray lines at a number of elevations to help fill in any gaps.

Figure 36: Empty spaces along the ray line can potentially cause image artefacts during reconstruction.

Fig. 36a demonstrates how a gap can sometimes be created when a group of voxels are be selected, meaning that the attenuation data for that x-ray line is spread across voxels in a less accurate way. This error may compound as the volume rotates through all of its projection angles. Fig. 36b demonstrates how tracing multiple ray lines per detector element can assist with this problem. When the circular volume geometry is eventually implemented on a MARS scanner, the relative size of the voxels will likely be significantly smaller than the size of the detector elements, so this "blind spot" problem may only be due to the fact that the ring thickness and detector height were kept equal for the initial implementation of these algorithms.

In terms of the projection angles, it is likely that using 720 angles for the reconstructions may have significantly contributed to the image artefacts. The circular volume geometry has a huge dependence on the number three, with three voxels in the centre ring and each successive ring containing six more voxels than the last. By using a number of projection angles that is also a multiple of three, inconsistencies with the voxel intersection function may have occurred over and over again due to the repeating nature of the step in rotation angle.

It seems that a reconstruction algorithm must be developed with the circular volume geometry specifically in mind. The system matrix will need to include details such as weighting factors to facilitate a more accurate reconstruction. Integrating the circular volume geometry into an ART-based reconstruction algorithm should be much simpler, because it doesn't rely on the procedure of smearing projection data across a string of voxels.

6.7 CONCLUSION

This chapter presented methods for displaying the circular volume geometry on a computer screen, and some preliminary analysis of how effective it is to plug-in the circular volume geometry into a well-known reconstruction algorithm that expects to be using traditional square voxels. The software development and analysis of the circular volume geometry represents a significant contribution to the development of the new iterative reconstruction method being developed by the MARS group.

Two voxel mapping functions were created, called VoxelMapFun and PointMapFun. The latter was significantly faster and also more highly parallelisable, and was subsequently developed further into C++ and CUDA functions. The functions covered the entire volume with Cartesian points and determined which points lay within individual voxels. A look-up table was generated that stored this information. The entire process of creating the look-up tables and drawing the volume was very fast and only took a few seconds. By comparing the speeds of all implementations, it was found that the CUDA version was the fastest. This wasn't a surprising result, since the algorithm employs data-based parallelism and is highly memory-efficient.

Making use of the new visualisation functions, the circular volume geometry was integrated into a FBP algorithm in an attempt to compare it against reconstructions that employ traditional square voxels. The preliminary results indicate that a lot more work is needed to be done to achieve image quality parity between the two geometric models, and the reconstruction method itself will need to be developed with some knowledge of the circular volume geometry. Integrating the circular volume geometry into an ART-based reconstruction algorithm will be much simpler and produce better quality images.

- There are several techniques that can transform a circular volume geometry into a square geometry, including mapping Cartesian points and using bi-linear interpolation.
- A grid mapping transformation was implemented in two algorithms, with the PointMapFun version being superior.
- The CUDA implementation of PointMapFun was found to be significantly faster than the other implementations, due to efficient memory usage and the highly parallel nature of the algorithm.
- The circular volume geometry cannot be simply plugged in to an existing FBP reconstruction algorithm without further modifications. Integrating it into an ART-based algorithm will result in better quality images.

7

CONCLUSION

The work completed during this thesis has made a significant contribution to research currently being conducted by members of the MARS group, and many of the algorithms and concepts developed will have an impact on the new iterative reconstruction algorithm currently in development. The key contributions to MARS MD and the cylindrical volume geometry are described in the following sections.

7.1 MARS MD IMPROVEMENTS

The original MARS MD software ran as a parallel process on the Blue Fern supercomputer, but took several hours to perform material decomposition on a standard set of CT images. As part of this thesis several improvements were made to the software. The main changes were: moving the code base from GNU Octave to MATLAB; using more appropriate data structures; vectorising and then parallelising the three core functions with MATLAB `parfor`-loops. In addition to performance improvements, a new graphical user interface was designed in MATLAB which allowed the user to interact with MARS MD more quickly and easily.

The improvements made to the MARS MD software resulted in computation time being reduced from hours to minutes. Combining this performance boost with the new user interface, the software ultimately became significantly more responsive and user-friendly. The new software has been used by several MARS team members as part of their research and has contributed to the publication of papers and theses.

7.2 CYLINDRICAL VOLUME GEOMETRY

The second part of this thesis involved implementing the geometric formulations described in previous chapters into computer algorithms, and analysing the subsequent performance and visual consistency after reconstruction. Two sets of algorithms were presented: (1) generating look-up tables to represent the system matrix of an iterative reconstruction algorithm; and (2) generating look-up tables to efficiently transform the two-dimensional circular geometry into a square pixel

geometry. All algorithms were coded first in MATLAB, followed by standard C++ and CUDA. The three implementations were compared in terms of performance, with the aim of determining the appropriate computing architecture to base future algorithms on.

The system matrix look-up tables that were generated were very large and ranged from several hundred megabytes up to a couple of gigabytes. The look-up tables were generated so that the system matrix could be computed before a reconstruction takes place and hence speed up reconstruction time. After performance testing, it was found that the standard C++ implementation was the fastest. This was due to the fact that the algorithm itself consisted of a significant number of branching statements which is not conducive to efficient GPU performance. The algorithm was implemented as task-based parallelism, which GPUs are not designed for. Additionally, the amount of temporary memory required during computation was large, which mean that only a minimal number of parallel threads could be computed concurrently. The result indicated that future versions of this algorithm would be best suited to run as a parallel process on a multi-core CPU. The next step in development of the system matrix algorithm will implement the remainder of the cylindrical volume geometry formulation which eliminates the angular dependence from the equations. This change will provide a reduction in the computational complexity and look-up table size of the order of 10^3.

The visualisation of the circular volume geometry was implemented in a similar way. Of the two algorithms developed, PointMapFun was significantly faster and had a higher degree of inherent parallelism. After being prototyped in MATLAB it was then extended into C++ and CUDA. After performance testing, the CUDA implementation was shown to be significantly faster than the standard C++ implementation. This was due to the overall simplicity of the algorithm and the fact that it employed data-based parallelism. The result indicated that future versions of the algorithm should be split into parallel processes on a GPU and computed on the fly.

Finally, the circular volume visualisation methods was employed to display the results of FBP reconstruction tests. This analytical step was undertaken to see if the circular volume geometry could be easily integrated into an existing reconstruction algorithm that relies upon square voxels. The resulting image artefacts highlighted the importance of choosing appropriate projection angles and angle step sizes, but integration into an ART-based algorithm will be much more effective.

[1] Marco Jacobs. Ten luminary quotes on parallel programming. http://www.vectorfabrics.com/blog/item/parallel_programming_quotes, October 2011. Accessed: 28-06-2014.

[2] Rafidah Binti Zainon. *Spectral micro-CT imaging of ex vivo atherosclerotic plaque*. PhD thesis, Christchurch, 2012.

[3] Christopher James Bateman. *Methods for Material Discrimination in MARS Multi-Energy CT*. PhD thesis, Christchurch, 2015.

[4] Eric W Weisstein. Circular Segment. http://mathworld.wolfram.com/CircularSegment.html. Accessed: 15-08-14.

[5] Jerrold T Bushberg. *The essential physics of medical imaging*. Wolters Kluwer Health/Lippincott Williams & Wilkins, Philadelphia, 2012.

[6] A Eklund, P Dufort, D Forsberg, and S M LaConte. Medical image processing on the GPU - Past, present and future. *Medical Image Analysis*, 17(8):1073–1094, 2013.

[7] Niels Johannes Antonius de Ruiter. GPU Accelerated Intermixing as a Framework for Interactively Visualizing Spectral CT Data. Master's thesis, University of Canterbury. Centre of Bioengineering, 2011.

[8] M J Rodríguez-Alvarez, A Soriano, A Iborra, F Sánchez, A J González, P Conde, L Hernández, L Moliner, A Orero, L F Vidal, and J M Benlloch. Expectation maximization (EM) algorithms using polar symmetries for computed tomography (CT) image reconstruction. *Computers in Biology and Medicine*, 43(8):1053–1061, 2013.

[9] R Ballabriga, M Campbell, E H M Heijne, X Llopart, and L Tlustos. The Medipix3 prototype, a pixel readout chip working in single photon counting mode with improved spectrometric performance. *IEEE Transactions on Nuclear Science*, 54(5):1824–1829, 2007.

[10] M A Hurrell, A P H Butler, N J Cook, P H Butler, J P Ronaldson, and R Zainon. Spectral Hounsfield units: A new radiological concept. *European Radiology*, 22(5):1008–1013, 2012.

[11] Wolfgang Birkfellner. *Applied medical image processing: a basic course*. CRC Press/Taylor & Francis, Boca Raton, 2011.

[12] Liubov A Flores, Vicent Vidal, Patricia Mayo, Francisco Rodenas, and Gumersindo Verdú. Parallel CT image reconstruction based on GPUs. *Proceedings of the 12th International Symposium on Radiation Physics (ISRP 2012)*, 95(0):247–250, February 2014.

[13] J Cheng, M Grossman, and T McKercher. *Professional CUDA C Programming*. EBL-Schweitzer. Wiley, 2014.

[14] The OpenMP® API Specification for Parallel Programming. `http://www.openmp.org`. Accessed: 02-02-2015.

[15] S Cook. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Applications of GPU computing series. Morgan Kaufmann, 2013.

[16] Wen-mei Hwu and David Kirk. *Programming massively parallel processors: a hands-on approach*. Elsevier/Morgan Kaufmann, Amsterdam, 2013.

[17] R Mafi and S Sirouspour. GPU-based acceleration of computations in nonlinear finite element deformation analysis. *International Journal for Numerical Methods in Biomedical Engineering*, 30(3):365–381, 2014.

[18] Intel® Xeon Phi™ Product Family. `http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html`, 2014. Accessed: 05-06-2014.

[19] John W Eaton, David Bateman, and Søren Hauberg. *GNU Octave version 3.0.1 manual: a high-level interactive language for numerical computations*. CreateSpace Independent Publishing Platform, 2009.

[20] R E Alvarez and A MacOvski. Energy-selective reconstructions in X-ray computerised tomography. *Physics in Medicine and Biology*, 21(5):733–744, 1976.

[21] H Bornefalk. XCOM intrinsic dimensionality for low-Z elements at diagnostic energies. *Medical Physics*, 39(2):654–657, 2012.

[22] H Q Le and S Molloi. Segmentation and quantification of materials with energy discriminating computed tomography: A phantom study. *Medical Physics*, 38(1):228–237, 2011.

[23] A M Alessio and L R MacDonald. Quantitative material characterization from multi-energy photon counting CT. *Medical Physics*, 40(3), 2013.

[24] X Liu, L Yu, A N Primak, and C H McCollough. Quantitative imaging of element composition and mass fraction using dual-energy CT: Three-material decomposition. *Medical Physics*, 36 (5):1602–1609, 2009.

[25] J Wang, N Garg, X Duan, Y Liu, S Leng, L Yu, E L Ritman, B Kantor, and C H McCollough. Quantification of iron in the presence of calcium with dual-energy computed tomography (DECT) in an ex vivo porcine plaque model. *Physics in Medicine and Biology*, 56(22):7305–7316, 2011.

[26] *Parallel Computing Toolbox™ User's Guide*. The MathWorks, Inc, 2014.

[27] R Aamir, A Chernoglazov, C J Bateman, A P H Butler, P H Butler, N G Anderson, S T Bell, R K Panta, J L Healy, J L Mohr, K Rajendran, M F Walsh, N De Ruiter, S P Gieseg, T Woodfield, P F Renaud, L Brooke, S Abdul-Majid, M Clyne, R Glendenning, P J Bones, M Billinghurst, C Bartneck, H Mandalika, R Grasset, N Schleich, N Scott, S J Nik, A Opie, T Janmale, D N Tang, D Kim, R M Doesburg, R Zainon, J P Ronaldson, N J Cook, D J Smithies, and K Hodge. MARS spectral molecular imaging of lamb tissue: Data collection and image analysis. *Journal of Instrumentation*, 9(2), 2014.

[28] María-José Rodríguez-Alvarez, Filomeno Sánchez, Antonio Soriano, Amadeo Iborra, and Cibeles Mora. Exploiting symmetries for weight matrix design in CT imaging. *Mathematical models of addictive behaviour, medicine & engineering*, 54(7–8):1655–1664, October 2011.

[29] L Jian, L Litao, C Peng, S Qi, and W Zhifang. Rotating polar-coordinate ART applied in industrial CT image reconstruction. *NDT and E International*, 40(4):333–336, 2007.

[30] Erich Hartmann. Geometry and Algorithms for Computer Aided Design. Technical report, Darmstadt University of Technology, February 2004.

[31] Junjun Xin, Chuck Bardel, Lalita Udpa, and Satish Udpa. GPU Implementation of Simultaneous Iterative Reconstruction Techniques for Computed Tomography. *The 39th Annual Review of Progress in Quantitative Nondestructive Evaluation, VOLS 32A & 32B*, 1511:777–784, 2013.

[32] Xun Jia, Yifei Lou, Ruijiang Li, William Y Song, and Steve B Jiang. GPU-based fast cone beam CT reconstruction from undersampled and noisy projection data via total variation. *Medical Physics*, 37(4):1757–1760, 2010.

[33] Xing Zhao, Jing-Jing Hu, and Tao Yang. GPU based iterative cone-beam CT reconstruction using empty space skipping technique. *Journal of X-Ray Science & Technology*, 21(1):53–69, March 2013.

[34] Yining Zhu, Yunsong Zhao, and Xing Zhao. A multi-thread scheduling method for 3D CT image reconstruction using multi-GPU. *Journal of X-Ray Science & Technology*, 20(2):187–197, June 2012.

[35] Mark Bangert. CT reconstruction package. `http://www.mathworks.com/matlabcentral/fileexchange/34608-ct-reconstruction-package`, January 2012. Accessed: 22-10-14.

[36] I A Feldkamp, L C Davis, and J W Kress. Practical Cone-Beam Algorithm. *Journal of the Optical Society of America A: Optics and Image Science, and Vision*, 1(6):612–619, 1984.